CRANFIELD UNIVERSITY

CARL MYHILL

FACILITATING THE COMPREHENSION OF HUMAN-COMPUTER INTERACTION DESIGN INTENT WITHIN A SOFTWARE TEAM

HUMAN FACTORS TECHNOLOGY GROUP
COLLEGE OF AERONAUTICS

PhD THESIS

CRANFIELD UNIVERSITY

HUMAN FACTORS TECHNOLOGY GROUP
COLLEGE OF AERONAUTICS

PhD THESIS

Academic Year 1998-9

CARL MYHILL

Facilitating the Comprehension of Human-Computer Interaction Design Intent within a Software Team

Supervisor: Dr. Peter Brooks

December 1998

Abstract

A large proportion of today's software development is unsuccessful. One reason for this is thought to be lack of attention to the user. Maintaining a user-centred focus during software production is regarded as a major problem. Introducing an HCI designer role into the software team (they usually function as external advisors) is thought to be a means of addressing this problem.

Issues surrounding the introduction of an HCI designer role into software teams were explored by a qualitative investigation. Participant-observation studies were carried out on two year-long software projects, with the researcher performing the role of HCI designer within the software teams. Aspects of comprehension within the team were found to be fundamental to successful collaboration. Prototypes were found to be an effective means of facilitating team members' comprehension of HCI design intent, and of maintaining conceptual integrity. However, this use of prototypes was flawed because they introduced the potential for ambiguity and they were inaccessible.

Focusing on the collaboration of the HCI designer and programmers, requirements for a prototype-centred explanation tool were specified to exploit the potential of prototyping to facilitate comprehension, by addressing the flaws discovered. Such a tool, called 'ProtoTour', was designed and implemented, based on the requirements specified.

An experiment was conducted with 22 commercial programmers to ascertain whether a ProtoTour representation of an existing, commercially developed prototype, facilitated comprehension more effectively and was more accessible than a conventional prototype. Results of the experiment found that programmers using ProtoTour gained a significantly better understanding of HCI design intent, than programmers using a conventional prototype. Those using ProtoTour also asked the HCI designer significantly fewer questions about the HCI design intent. Results suggest that prototype-centred explanation tools have the potential to improve programmers' comprehension of HCI design intent.

Introducing an HCI designer into a software team was found to be an effective way of improving the user-centred focus of software during production. A prototype-centred explanation tool appears to have potential as a means of helping programmers comprehend HCI design intent.

Table of Contents

List of Figures

Figure

List of Tables

Table

Chapter 1 Literature Review Relating to Software Production

1.1 Introduction

Research into software production covers a vast range of topics and a diversity of software production contexts. From the accompanying literature, it is not always clear what the software production context is for the research topic. Many researchers make general claims from their findings in a particular software production context. This is not always appropriate as there are several clearly distinguishable types of software production context. Grudin (1996) is careful to classify research findings into four broad categories of software production contexts: off-the-shelf product development, in-house development, competitively bid contract development (large formal contracts), and customised software development (smaller less formal contracts). In a study of the design process representative of the UK environment, Harker (1991) considered the main distinction to be between off-the-shelf products and bespoke software.

Unfortunately, not enough researchers are clear about the software production context to which their work applies. Consequently, distinguishing between research applicable to one context or another could be seen as a research project on its own. Whilst it is not the purpose of this literature review to categorise all previous research on software production, it is deliberately focussed on the context of the research area of bespoke software.

Scope of Literature Review

The literature encompasses material that pertains to the business of producing software in the commercial world. Diverse sources have been utilised in this research but the coverage of this review has been designed to provide insight into software production specifically.

It is important to note that this research is in the interdisciplinary field of Human-Computer Interaction. This explains both the diversity of sources included and the reason for not focussing on literature relating to only one of the particular disciplines (e.g. psychology).

1.2 The End Product

One way to begin investigating current software production is to examine the end product of this process. Section 1.2.1 describes some aspects of the failure of software technology to either reach the marketplace or be accepted by users. This is followed by an overview of researchers' views on the current state of user interfaces.

1.2.1 The Failure of Software Technology

Many researchers have shown concern about the lack of success of a large proportion of software ventures. Browne (1994) claimed that over the last 30 years computers have failed to improve productivity even on the most generous productivity measures. A study cited by Land, Le Quesne and Wijegunaratne (1991) showed that only 1% of large systems (defined as those having more than 50,000 lines of code) met users' needs and were finished on time within budget. Land *et al.* stated that the average large system is a year late and costs twice the original estimate. A study of new products evaluated their likelihood of failure to be between 66 and 90% (Business Week, 1992). Gladden (1982) cited a study which quoted 75% of software development was never completed or was not used if completed; Grudin (1993) referred to frequent product failure and dissatisfaction. A more recent study by Grudin (1996) suggests that reliable figures are hard to find, which is probably the reality of the situation. He cited an informal poll of several experienced development managers from a number of companies, which estimated completion rates between 10 and 50%. Some experienced developers reported to never having seen a project through to completion, he stated.

Grudin (1996) assessed the likelihood of failure of the different broad categories of software production contexts. He found that completion rates were better for systems built under contract than under other production contexts, as both the client and the provider had most to lose in this context. In his opinion, the high completion rate of contract developments explained the absence of project mortality considerations in the literature. He found that small projects developed under contract for specific customers in a particular market can be even more likely to succeed, because the software producer is more likely to continue working with the client after implementation. However, Grudin's beliefs are not universal. Martin (1988) reported that competitive bid contract systems are often unusable on delivery without further work.

In HCI and Human Factors literature the attribution of blame for these failures is unsurprising. Gould (1988) cited a study of 12 major business failures, all of which were contributed to by a lack of understanding of the "business", "application" or "user set". Kearney (1990) believed the key failure reason to be the neglect of human and organisational factors. Similarly, Baeker and Buxton (1987a) similarly suggested that the user interface is often the single most important factor in determining the success or failure of a system. Wasserman (1987) concurred that the user interface is often the principle determinant of a system's success.

Other reasons for the failure of software technology include to aspects of software project management, such as poor estimating and monitoring plus a lack of understanding of software project's nature. Other researchers have attributed the problem to inadequate tools and methods for system design and construction (Land *et al.*, 1991).

1.2.2 The User Interface

The goal of HCI and Human Factors practitioners is to empower the user (Laurel, 1990). Many researchers believe that this goal is not being reached. Norman (1990) illustrates this viewpoint:

"*What is the good news about computers and their interface? Alan Kay is reported to have said that "the Macintosh has the first interface good enough to be criticised." That is supposed to be the good news? Sorry folks, things are seriously wrong in interface land...*" (p. 209)

A similar view was expressed by Nelson (1990):

"I am dismayed at the dreariness of the interactive software that people today think is liberating and forward-looking. Compared to what it should and will be, today's interactive software is wooden, obtuse, clumsy, and confused. The pervasive lack of imagination and good design is appalling." (p. 235)

These are not up to the minute views, but since their publication nothing has revolutionised user interfaces. More recently, Norman (1996) has stated that he believes 'machines' have become too complex. Norman has recently reaffirmed his attribution of this complexity to the "accident" which he described in 1990 as the adaption of general purpose technology using very general tools for very specialised tasks.

Like Norman and Nelson, Grudin (1993) has also stated the view that the usability and utility of software could be markedly improved. Other researchers provide a reminder that historically the user interface and usability were considered an afterthought (Wasserman, 1987; Whiteside, Bennett and Holtzblatt, 1988). Although a proportion of developers today have broken away from this attitude, it should be remembered that many current software developers began their careers when the user interface was strictly an afterthought.

One indisputable fact is that the user interface has now become a more significant part of software development. In a survey of 74 developers (attending a human factors conference, hence the positively weighted results), it was found that an average of 48% of code was devoted to the interface (Myers and Rosson, 1992). They further found that 45% of time in the design phase was devoted to the user interface, 50% during the implementation phase and 37% during the maintenance phase. Baeker and Buxton (1987a) cited several studies, which demonstrated that between 30 and 59% of the operational code supported the user interface, making it one of the most expensive elements of the code. Heckel (1991) cited a study which showed that the quality of the interface is three times as important as structured programming to the success of a software project.

Part of the reason for user interfaces taking on more significance is thought to be an increase in complexity, particularly due to the dominance of windows-based Graphical User Interfaces (GUIs). This complexity is evidenced by the following views. Myers and Rosson (1992) highlighted the fact that this complexity makes the programming task more difficult and adds further complexity through necessitating iterative design. According to Carey, McKerlie, Bubie and Wilson (1991), the evolution of user interfaces has increased design complexity by increasing the available design options. Mantei and Teorey (1988) described user interface design as "a nightmare of detailed decisions". Erickson (1990) believed user interface design to be complex for three reasons: firstly, it is hard to come up with solutions; secondly, designing the user interface is a compromise of competing "desiderata"; and thirdly, user interface design is interdisciplinary and political. Grudin (1993) suggested that further complications occur when division of labour divides the responsibility for the user interface. Gould, Boies, Levy, Richards, and Schoonard (1990) held a similar view and suggested that only one person should be responsible for the user interface.

Some researchers think that there is too much emphasis on the 'user interface' when research should focus on how people can best be supported in their work (Bannon and Bødker, 1991; Norman, 1990). Maxwell (1996) showed concern that the term 'user interface' reinforces a model which treats the system purely as the functional and interactive software and the user as a separate entity. Whilst these views clearly have merit, they are difficult to reconcile with attitudes prevalent in modern day software development.

Baeker and Buxton (1987a) believed the user interface to be one of the most poorly understood aspects of any system, with its success or failure determined by a complex range of poorly comprehended and subtly interrelated issues. Although there are good reasons why user interface design will always be an aspect of the system which is complex and difficult to understand, it is interesting to note when eminent software engineers began to recognise its importance. Brooks (1975) eloquently described user interface design in 'The Mythical-Man Month':

"By the <u>architecture</u> of a system, I mean the complete and detailed specification of the user interface…The architect of a system, like the architect of a building, is the user's agent. It is his job to bring professional and technical knowledge to bear in the unalloyed interest of the user, as opposed to the interests of the salesman, the fabricator, etc." (Brooks, 1975; p. 45)

It is perhaps surprising that the software engineering community failed to take Brooks' view seriously, and research emphasis on the user interface failed to gain ground until many years later.

1.2.3 Summary and Final Commentary

It is possible that failure figures are distorted as researchers strive to justify their work or add weight to its importance. Whether the failure statistics for software technology paint an accurate picture or not, it is clear that much new software technology does suffer from problems during production and acceptance. Despite this, software production is at the core of a thriving industry. Grudin's assessment appears the most credible. He expressed the view that project failure is determined by the type of software being produced, and viewed software developed under contract with external suppliers to have the best chance of survival. One source of blame for software failure is the neglect of people and organisational factors. The complexity of dealing with such issues during software production gives cause to take such a view seriously.

Critics believe that judging by existing user interfaces there is still a long way to go to improve user interface design. It does not take a user interface expert to realise that a large proportion of software products[1] reaching the desktop suffer from bad design to various degrees. However, it is a fact that the user interface now requires a large proportion of production effort and code to produce. This increase in the dominance of the user interface probably has more to do with the inherent complexity of programming GUIs and windowing software, than greater emphasis on user and task considerations. Although the user interface is less an afterthought now, this is partly due to its technical complexity and dominance rather than a change in emphasis towards user and task considerations. Many people involved in software production spent their formative years programming software where the user interface was an afterthought – some of these old ideas linger on. This may well be a fundamental reason why some researchers now object to the term 'user interface', claiming that the focus is wrong. These researchers would instead prefer to focus efforts on giving more consideration to the user and their job. A final observation of this section is that the importance of user interface design was identified as early as 1975 by an eminent member of the software engineering community, Brooks, but failed to become a serious issue in software engineering until many years later.

[1] This criticism also applies to many products of modern design such as video recorders, washing machines, microwave ovens, etc. (see Norman's (1988) 'The Psychology of Everyday Things' for more examples)

1.3 The Software Production Process

This section describes the dominant waterfall lifecycle, which underlies the majority of software development processes. The main alternative software production philosophy is iterative development so this is considered next. Other alternative approaches are discussed briefly.

1.3.1 The Waterfall Lifecycle

The waterfall lifecycle is a linear software production process where the deliverables from one phase of the process feed into the next (without overlap). For example, when the requirements phase is completed it is usual for this to feed forward to the design phase in the form of a written specification document. The waterfall lifecycle has been around for many years (Brooks (1995) has recently suggested that some of the chapters in his 1975 'Mythical Man-Month' were "tainted" by it). Usually credited to have come about from US defence and government contract projects, the waterfall lifecycle corresponds to what many people regard as the traditional view of software engineering (Benyon, 1995).

Boehm (1988) described the waterfall lifecycle as "dominant". The waterfall cycle was cited by Harker (1991) as the underlying philosophy of large scale bespoke development. Lansdale and Ormerod (1994) suggested it was the most "common approach" to development and Gordon and Bieman (1994) claimed that it "remains the dominant paradigm". Articles criticising the waterfall lifecycle have been published by respected researchers for many years, a recent example of which is Stephens (1996). Although probably not commonly followed in its pure form, **the underlying philosophy of the waterfall lifecycle is still prevalent in software development**.

A tide of opinion argues that the waterfall lifecycle does not work (e.g., Boehm, 1988; Stephens, 1996). Some of the reasons cited for this failure are that:

- it does not facilitate iterative design or development (Cockton, 1991; Grudin, 1991);

- it assumes that specifications can be complete and correct first time (see section 1.6);

- it emphasises written specifications and contracts (Grudin, 1991) which imposes demands on representations that available techniques cannot meet (Benyon, 1995);

- it fails to manage inevitable changes in requirements (McCracken and Jackson, 1981);

- it does not support user involvement in software production (McCracken and Jackson, 1981); in fact, Grudin (1991) states that it presents a "formidable barrier" to user involvement;

- it bears little resemblance to how designers go about their work (Benyon, 1995);
- it fails to treat software as a problem solving process (Curtis, Krasner, Shen and Iscoe, 1987);
- it fails to acknowledge the "vast communication" that is required for team members to share understanding (Curtis *et al.*, 1987).

Of all the cited reasons for its failure, the waterfall's inability to facilitate iteration and change is the major one. The waterfall seems to be founded on a basic misunderstanding of the nature of software production.

If the waterfall is so bad, it is germane to ask why it survives. The main is that it presents a realistic face to the business world. It is a natural fit to contract developments (Grudin, 1991). Fixed price contracts are an accepted way of doing business in software, even software produced in-house is likely to be subject to a contract in the form of a fixed budget. Although largely unrealistic, the waterfall lifecycle at least provides *some* way of estimating the resources required, duration and cost of a proposed software development. The reality of software production is that because of the large number of unknowns, it is often completely unrealistic to estimate how long development will take. Unfortunately the reality of the situation does not conform to the commercial context in which software projects are carried out. Although there are advocates of project managers sticking to their guns and, as suggested by McCarthy (1995), rather than giving a "bogus date" having the courage to say "we don't know when the software will be done", few project managers would be in a strong enough position to take this more realistic stand (McCarthy works for Microsoft who recently launched Windows 95 in 1996).

Another major reason why the waterfall survives is that it is manageable and helpful in controlling large software projects (Benyon, 1995; Curtis *et al.*, 1987). Curtis *et al.* suggested that the waterfall serves the need for management accountability. It clearly does take steps in this direction with an emphasis on the achievement of milestones. These facilitate project management and monitoring, providing an indication of progress to management and clients.

Norman (1996) suggests that the way industry is structured does not lend itself to the process of design. Moving away from the waterfall philosophy would require restructuring the way that software procurement is carried out. The major element of this restructuring would be the negotiation of flexible contracts (see later notes on RAD - section 1.3.2), examples of which are beginning to emerge. The following statement from Grudin (1993) about obstacles to participatory design in large product development organisations also sums up the general situation quite well:

"Eventually organisational change may be required to overcome the constraints and forces confronting developers. In the meantime, those working with such organisations [large product development organisations] must be aware of the problems and seek constructive paths around them." (Grudin, 1993; p. 99)

Several researchers have now tacitly acknowledged the continuing dominance of the waterfall lifecycle and have packaged their work in a format which readily links with it (see section 1.8.4).

1.3.2 An Alternative to the Waterfall Lifecycle: Iterative Development

The major alternative to the waterfall lifecycle is iterative development (sometimes called incremental build, an empirical approach, evolutionary development or evolutionary prototyping). The underlying philosophy of iterative development is to produce and deliver the software in a number of stages of increasing functionality. The aim is to provide the user with software that they can actually use very early on in the development process. As each new installment of software is delivered, various forms of testing can be carried out and requirements for future installments confirmed. This type of approach clearly has some implicit user-centred advantages when compared with **the waterfall approach which assumes that requirements can all be captured and accurately specified during the early stages of the project and the software delivered in one large installment at the very end of the project**.

A large number of researchers have suggested that an iterative approach is necessary for software production (e.g., Stephen 1996, Benyon, 1995). It was suggested by Brooks (1977) that the empirical approach is the only way to build reliable computer systems. Later (1986), he suggested that we need to grow software organically. Whiteside *et al.* (1988) concurred with these views. Gabriel (1994) also believed that the waterfall's goal of total design before manufacture was the wrong approach to software production and incremental development would be right. Olson, Buxton, Ehrich, Kasik, Rhyne and Sibert (1990) specifically suggested that the user interface development process is, of necessity, an iterative process. Cockton (1991) argued that current methodologies incorporating iteration are increasing their potential compatibility with best HCI design and evaluation practice. This is quite likely because, as Lansdale and Ormerod (1994) stated, any approach which makes you think of the user is likely to provide insight, and many iterative process do involve the user implicitly.

In many respects iterative development is better matched with the realities of software projects with its high levels of uncertainty and complexity, which is impossible to foresee when writing specifications early in a waterfall process (see section 1.6). However, the use of iterative software production processes has not become as widespread as would be expected given the amount of research claiming the waterfall does not work and that which claims iterative development does work.

In a somewhat biased survey of 40 software projects (biased because many of the 74 respondents were approached through human factors journals), Myers and Rosson (1992) found that:

"Many (42%) indicated that the work had been very evolutionary in nature, with design and implementation of the user interface proceeding in parallel (intertwined)."(p. 7)

This figure appears to be too high, this is obviously partly because of the sample bias but is perhaps also to do with the definition of 'evolutionary in nature' which Myers and Rosson used. It would not be uncommon for many projects ostensibly following a waterfall lifecycle to have elements of design and implementation proceeding in parallel, this would not make the underlying process 'evolutionary in nature'. Furthermore, if iterative development has really taken off, why is there still so much discussion of it (e.g., Stephens, 1996). However, even though figures reported by Myers and Rosson seem to overstate the position, other evidence suggests that iterative development is gaining ground in certain situations.

As Stephens (1996) and McCarthy (1995) have stated, Microsoft uses an iterative development process. Microsoft's beta testing is now almost legendary. It is unlikely, however, that many software development companies would be able to arrange such extensive testing with such a willing user base. It is Microsoft's unique position in the industry which allows it to iteratively develop and consequently be vague about product delivery dates.

An iterative development approach called Rapid Application Development (RAD) has gained some ground recently in certain development contexts in the UK. Although RAD is not a new idea (some 20 years old), software development technologies which facilitate the approach have gained acceptance over the last few years. Leaders among these technologies include Microsoft's Visual Basic and Borland's Delphi, both of which provide a far more accessible way of programming the complexities of modern GUIs than were previously available. RAD advocates are careful to point out that it is only suitable for certain types of projects and teams, and situations where there is a close supplier-client relationship (Stapleton, 1996). One broad category of suitable projects is therefore in-house software development. RAD also demands a flexible approach to contracts which are usually renegotiated every time a new software increment is delivered. At this point the client is able to re-assess whether the next software increment is worth paying for.

Opponents of the RAD approach suggest that it encourages 'hacking' (a colloquial expression for producing software in an ill-structured way without prior design). Stapleton (1996) strongly denied this and claimed that code produced by RAD is of higher quality than that produced by traditional processes (i.e. the waterfall). She cited a study at IBM which found RAD projects have lower maintenance costs than traditionally developed software. Stapleton claimed that lower maintenance costs show that RAD code is of higher than average quality. However, this is a very weak argument. RAD projects actively demand user involvement whereas traditional processes do not. It is far more likely that the reduced maintenance costs found in RAD projects are due to user involvement, which was largely absent from the traditionally developed software. This involvement enabled the RAD projects to achieve a better understanding of the real requirements. If requirements are right then maintenance will not be necessary and the quality of the RAD produced code will not even be apparent.

The fundamental barrier to iterative development is fixed contracts, which according to Grønbæk, Grudin, Bødker, and Bannon (1993) hinder iterative design regardless of the type of project considered. They stated that customers prefer the security of having a fixed price/fixed time contract, but the reality is that such contracts often “live in fiction”. At the software procurement stage, popular business practice requires a price and delivery date and is often far less receptive to a piecemeal development approach. The end result of software developed to a fixed price/fixed time contract may well betray the fact that such contracts are fiction, but this occurs a long time after contracts are signed. This commercial reality illustrates the frustration which is felt by researchers in this area. For example, Grudin (1993) believed that many successful systems in existence are the result of an undesirably long evolutionary process and recognised that the difficulty of developing such software is substantial.

1.3.3 Other Alternatives to the Waterfall Lifecycle

Many alternative development lifecycles have been created, although the indicates that few have caught on. This may be partially due to the fact that some major companies invest in developing their own lifecycle which they keep a closely guarded secret.

Some researchers (e.g. Grudin 1993) believe that Boehm’s (1988) Spiral Model is a credible alternative to the waterfall lifecycle. This model uses an expanding spiral to illustrate project progress and cumulative cost incurred. On each spiral revolution several project activities are revisited, for example, prototypes are constructed (which may evolve through the spiral into the final implementation) and a thorough risk assessment is carried out. The Spiral Model is designed to facilitate a flexible and evolutionary approach, combined with manageability aspects equal to the waterfall lifecycle. However, as Grudin (1993) stated, it is not widely used.

Benyon (1995) cited the Star lifecycle as a convincing model of the design process, but claimed that it does not facilitate project management. This would appear to be an accurate view of the Star lifecycle which was developed following detailed analysis of design practice (Preece, Rogers, Sharp, Benyon, Holland, and Carey, 1994). The Star model represents design activities (e.g. prototyping and implementation) as nodes on a star with evaluation as a central activity linked to all nodes. The model is not prescriptive about the order that tasks are carried out in. This may map the way designers actually work, but it makes it difficult to manage.

Some researchers (e.g., Agresti, 1986) suggest that prototyping is an alternative to the waterfall lifecycle, but this would seem to be both an overstatement of what prototyping is and an understatement of what is required of a production process.

Grudin’s (1993) suggestion that **an innovative process is less likely to be as acceptable to management as a written specification** summarises the situation well (written specifications being a fundamental part of the waterfall lifecycle).

1.3.4 Summary and Final Commentary

The waterfall lifecycle is widely believed to be inappropriate to the realities of the software production iteratively. However, key assumptions of the waterfall lifecycle allow projects to be planned and costed during their early stages. These approximate plans and costings are acceptable to business as they provide a means to procure software at a fixed cost/time. Unfortunately the nature of software production, normally involving a large number of unknowns, often means that plans and costings produced early in the project are inaccurate. The reality of the situation is that many current development processes retain waterfall lifecycle underpinning, because it facilitates cost estimation and effort (however inaccurate). Other approaches, such as iterative development, are widely believed to be far more appropriate, but they have an honest underpinning which acknowledges the unknowns and resists making fallible cost and effort estimates. The technical credibility of iterative development is overall outweighed by its lack of commercial credibility in a contract led industry. Alternative development processes are only succeeding in projects which meet a certain set of conditions. Therefore, particularly for software produced under contract, the waterfall lifecycle lives on, and as Grudin (1993) concluded,

"This strongly phased process is the reality to be addressed."(p. 103)

1.4 A Pragmatic View of Software Production

Theoretical processes for producing software deviate widely from what actually happens in practice. This section aims to provide an insight into what it is really like to produce software. The nature of the design process is described (see 1.4.1 & 2 which covers aspects of change affecting software production). The degree of user involvement in commercial software projects, practical issues surrounding the organisation of a software project, and several other important aspects of commercial software practice are described.

1.4.1 The Nature of Design Process

The design process has been described as eclectic and chaotic (Craig, 1991), inherently opportunistic (Carey *et al.*, 1991; Guindon, 1990) and ill-structured (Guindon, 1990). Rubenstein and Hersh (1987) believed that because design is an art as well as a science, it would never be a completely rational process. Rittel and Weber (1973) considered design to be a "wicked problem", that is, a problem whose formulation is necessarily vague and whose optimal solution cannot be practically found or measured. User interface design is often described as "ad hoc" (e.g., Due, Jorgensen and Nielsen, 1991; Stewart, 1991). Due *et al.* (1991) found that user interface design takes place in a "highly turbulent" organisational context with an extremely fragmented working situation. In a similar vein, Bødker, Grønbæk and Kyng (1993) outlined the political and conflict causing nature of the design process.

Harker (1991) discovered considerable diversity in activities across the design processes studied, which is consistent with the findings of Wilson, Bekker, Johnson, and Johnson (1996). They noted variations in design practice concerning user activities.

Clearly, design is not as straightforward an activity design process and methodology literature might indicate.

1.4.2 Change

One factor frequently ignored by software related literature is that software production is highly susceptible to change, some believe that such change is inherent in software projects (Beladay and Lehman, 1979; Brooks, 1986; Miller-Jacobs, 1991). Part of the reason for such change is that it is almost impossible to specify all requirements for the software at the outset of the project (see section 1.6). As people involved with the software production (users, clients, managers, user interface designers, programmers, project managers, etc) gain greater insight or visibility of the software, they will often modify their view of precisely what it should do. Curtis *et al.* (1987) described change arising from internal and external factors as "requirements volatility". The level of disturbance to the software production process that changing requirements causes depends on a multitude of factors. These range from how early on in the process the change was identified to the flexibility of the development team. Brooks (1986) believed that some of the pressure for change comes from the fact that it is inherently easy to change. Whilst this may account for some changes, difficulty visualising a proposed computer software and interpretation of abstractly expressed requirements are just two examples of equally likely causes.

1.4.3 User Involvement in Software Production

There is a variety of levels and styles of user involvement in software production reported in the literature. This seems likely to be linked with the broad characteristics of the software production itself. For example, Harker (1991) found there to be a clearer commitment to users in bespoke software development.

General surveys have been conducted into user involvement which do not distinguish between types of projects. Wilson *et al.* (1996) found that 60% of the 25 projects studied involved users and the study of 40 projects reported by Myers and Rosson (1992) showed that only 43% of software was tested with users. This leaves a disconcertingly large proportion of software being developed without user involvement. Of further concern is the fact that even on the projects with user involvement, Wilson *et al.* regarded the extent of access designers had to users and their workplace, to be seriously inadequate. The Myers and Rosson study also found that the most commonly cited problem by respondents was finding appropriate test subjects.

In software production it is very common for software requirements to come from either the management of the client organisation (Benyon, 1995; Bødker *et al.*, 1993; Grudin, 1993) or the marketing department of the supplier organisation (Harker, 1991; Curtis *et al.*, 1987). Preliminary software requirements rarely come from users. In product development contexts, Grudin (1993) found that users are not truly identified until development is complete and the product is marketed. Other development contexts are not so far removed from this condition. Grudin found that current practice precludes user involvement.

Another frustrating aspect of the reality of user involvement is highlighted by Karat (1996) who sarcastically described, “Enlightened organisations who have to have proof that you need to talk to the users”. Such companies, both client and software producing organisations, are not rare. For example, Grudin (1993) suggested that product development companies are not aware of the virtues of participatory design (i.e. full and explicit involvement of users in the software design process). This is a view echoed by designers in the 1996 Wilson *et al.* study. They reported obstacles arising from management viewing user involvement as an additional, optional activity, and marketing staff who thought they knew what the users wanted and consequently saw no need to involve the users.

One of the potentially most damaging elements of user involvement, which perpetuates bad design and prevents designers learning from their mistakes, is not widely recognised in the literature. Following the installation of completed software, the design team is often disbanded or moves on to another project. Field service teams are then used to support the users of the new software, which in effect shields developers from user feedback (Grudin, 1993). Grudin pointed out that developers are rarely aware of users' pain. This isolates developers, therefore they do not learn whether their design intuition is good or bad, or what future improvements are necessary.

1.4.4 Project Organisation

Brooks (1975) was probably one of the first to question what implementers should do whilst architects (user interface designers) were writing the specifications. This begins to illustrate what Newman (1991) later cited as a problem. Requirements analysis is usually performed in isolation from design by requirements analysts. The analysts make assumptions about the design, which the design team must understand in order to make sensible use of the requirements. If, as Button (1994) suggested, the determination of the requirements is purely a practical matter because the search for requirements is never ending, Newman's observation would lead to the situation where design teams cannot possibly obtain all relevant information from the requirements analysts.

The separation of development costs from maintenance costs (which can be considerable, between 60-90% of total project cost is cited by Grudin (1996)) removes accountability for maintenance from designers and developers. Grudin (1993) cited the general neglect of on-line help as an example of such divided responsibilities. He argued that a good help system can save a company a substantial sum in maintenance calls but the saving would probably be in the budget of a customer service department, while the effort and expense would have to come from the development budget. Hakiel (1995) cited a similarly illustrative anecdote of a project manager who was pleased to note that none of the recent service calls received were attributable to errors in the code, failing to appreciate the irony that he still had appreciable service organisation overheads. Thus, the project manager had not even conceived that the design of the software was probably instrumental in causing the maintenance costs.

1.4.5 Other Implementation Issues

There are potentially hundreds of other implementation issues which arise during the practice of software development, this section covers a few of the more common ones.

Tyldesley (1990) claimed that there will always be problems that effect usability which only arise during implementation. This comment was made in the context of development of office software which is clearly not at the highest end of the interactional complexity scale. Similarly, Brooks (1975) commented that no matter how precise the specification, as implementation proceeds, countless questions of architectural interpretation arise (Brooks defines architecture as external software appearance, i.e. user interface).

Harker (1991) believed that the tendency to underestimate the demands on development effort is endemic to all software development processes. If taken at face, this would suggest that tight deadlines are also endemic to all software development processes as personnel struggle to meet their ambitious estimates. This goes some way to explaining why some practitioners complain that usability is sacrificed to tight deadlines (Button, 1994; Trennor, 1995). Furthermore, tight deadlines pressurise activity towards the end of the project, such as testing. So while waterfall based lifecycles provide little enough scope for testing and feedback in theory, in a practical situation, even that testing may be thinned down. Grudin (1993) pointed out the irony of the situation when discussing the lack of late user involvement because the underlying code is frozen so that documentation can be completed. Concern that designers cannot "do all the tweaking and testing they advocate" when the program is not working a week before it is delivered has been expressed by Erickson (1990). Heckel (1991) concurred that software is unfriendly is that it is not tested in actual use.

1.4.6 Summary and Final Commentary

The software design process is chaotic in nature. User interface design is still often done in an ad hoc way. Because of the difficulties of fully grasping a complex problem and an abstract software solution, change is inherent in software projects. User involvement in software production practice is still alarming low. A study published in 1996 (Wilson *et al.*) stated that 40% of projects did not involve the user, while a 1992 publication quoted 57%. Whilst this is an improvement, 40% still represents a large proportion of software development lacking user involvement. The type of software is also thought to determine the likelihood of user involvement. It has been suggested that bespoke software developments often demonstrate a clearer commitment to users. It is still quite common for software requirements to emanate from management, rather than users. Furthermore, some organisations are seen to require some form of proof of the need to talk to users.

Because of the way software production is organised, when software is delivered, field service teams often support the users. Developers move on to other projects and are often never aware of the inadequacies of their software. Furthermore, the usual practice of separating the development from the maintenance budgets also serves to remove accountability from designers and developers.

Other difficult aspects of software development in practice include the likelihood that a number of problems affecting usability will not crop up until implementation. Finally, the tendency of software projects to be underestimated in terms of resources required is an indication of the tension which usually exists within a project. This also explains why there is often little time left at the end of a project to make changes following user trials.

1.5 Tools and Technology to Support Software Production

The use of tools and technology to support software production stems from the early days of software development when a bright programmer might develop a new debugger for other programmers to use. The evolution of tools has served programmers well, but the natural extension of useful individual desktop applications to Integrated Project Support Environments (IPSEs) for the whole team has not been so successful. Computer-Aided Software Engineering (CASE) tools are one step down from IPSEs and usually focus more on facilitating and controlling the software design process (internal design). Less ambitious individual tools for specification and design have also emerged. It is pertinent to investigate why so many of these tools fail.

1.5.1 The Failure of Tools

One major cause of failure is that the tools do not take account of the realities of the software production process and instead align themselves with the idealistic methodologies of software production. The nature of the software design process is chaotic and opportunistic (see section 1.4.1); tools failing to address this reality will be rejected.

Of all existing tools to support software production, IPSEs attempt to take the most control of the production process and the work of individuals. The structured techniques for software development that IPSEs support impose particular ways of working on the software team. They have to proceed with the work in a specific manner, plus they have to conform to methods of representing data and processes (Land *et al.*, 1991). It is therefore not surprising that the implementation of an IPSE in the project described by Land *et al.* was a failure. Other proponents of IPSEs concede that IPSE failure can be attributable to the administration burden, which it causes, being unacceptable to individuals in the team (Le Quesne, 1988).

CASE tools suffer from similar drawbacks to IPSEs. Harker (1991) surmised that the promise of CASE tools has not been realised. However, Hardy, Stobart, Thompson and Edwards (1995) reported surveys carried out in 1990 and 1994 which showed that CASE tools had become more widespread over this period.

A year-long assessment of design and specification tools was carried out by Curtis *et al.* (1987). They concluded that, although such tools have promise, they are generally far from “industrial strength”, actual applications of them were sparse and methods of using them, as yet undefined. They further expressed the concern that it was difficult to see how the use of such tools would scale up for use on large scale software projects.

Makers of tools to support the software production process often fail to recognise that it is a people-orientated process. Curtis *et al.* (1987) contended that if a process model does not represent the factors that control the largest share of the variability (i.e. people) in software production, it will not boost productivity and quality. The same could be said of software tools which fail to address this problem.

In the context of Computer Supported Collaborative Work (CSCW), Grudin (1988) cautioned that one chronic cause of failure in group work situations occurs when one group incurs more work, which benefits another. With respect to the development of tools, Grudin (1996) advised that these should be aimed at their use situations in the context of software development.

1.5.2 Absence of Tools

Some researchers cite the absence of tools as a means of justifying new tools. For example, Land *et al.* (1991) credit the evolution of IPSEs to inadequate design and construction tools. However, in some areas there does appear to be a genuine lack of tools. Primarily, those claiming to lack tools are people from other disciplines where an ability to program was lacking - thus they could not readily produce software tools for themselves. Damodaran (1991) cited the absence of tools to assist in user-centred deign to be a major obstacle to the progress of human factors in software production. As an interface designer at Apple, Wagner (1990) described her current tools as felt-tipped pens and drawing pads. She then described the tools she would require for her "perfect world" to carry out interface design more effectively. Tool requirements suggested revolve around the use of a flexible prototyping medium which is accessible to non-programmers.

1.5.3 Summary and Final Commentary

If prototyping is considered a tool, over the last decade it has probably had more influence on commercial software development practice than IPSEs, CASE, design, specification or any other tools. Prototyping is not covered here (see section 1.7).

Tools which attempt to structure have failed because they do not address the chaotic nature and people orientated nature of the software production process. The use of tools in software design and development does have promise if the tool makers become more realistic and begin to treat the software production process as more of a complex people problem. There are clearly still requirements for tools and many such requirements are from the non-programming community. The tradition of programmers using their intuition to develop tools for other programmers has to change.

1.6 Software Specification

Software specifications are a fundamental feature of the waterfall lifecycle which is believed to continue to dominate the software development industry (see section 1.3.1). Although different types of specifications can be distinguished, the basic purpose of a specification is to describe in detail, in written form, exactly what the proposed software should do. Such specifications are often written following a requirements phase of the development process where requirements analysts have thoroughly investigated the client requirements, described them and proposed a software solution. The specification is often regarded as those parts of the requirements phase that are deliverable and it usually signifies some changes to the software team (Browne, 1994; Vertelney and Booker, 1990). Often, requirements analysts move on to other projects and software designers and programmers are brought in to the team to begin designing and implementing the software. It is therefore clear that the specification carries an important communication burden within the changing software team.

Another important role of the specification is to act as a contract between software supplier and client. The specification usually defines the functionality of the software, which the project manager is obliged to deliver. Grudin (1991) believed that this reliance on specifications puts a wall between developers and users. Grønbæk *et al.* (1993) regarded competitive bid contracts based on early specifications to be unrealistic, believing in fact that fixed price and fixed time contracts often "live in fiction".

Although the specification is a very important and real aspect of today's software production, there is a large body of research which concludes that fully specifying software in advance of the development phase is wrong (Brooks, 1986; Gould, 1988; Miller-Jacobs, 1991; Swartout and Balzar, 1982). Brooks (1986) believed that conceptual structures are too complicated to be accurately specified in advance and too complex to be built faultlessly. Similarly, Curtis *et al.* (1987) found that important decisions made late in the development are not foreseen in the specification. Brooks (1975) also believed that countless questions arise during the implementation phase no matter how precise the specification.

From the perspective of user needs analysis, other researchers have suggested that fully specifying software in advance is not viable. Bødker (1991) and Ehn (1988) have found the assumption, that user needs can be completely analysed during the early project phases, leads to the production of software which is rejected by users. In the same vein, Bannon and Bødker (1991) claimed that it is never possible to get a full description of tasks or predict future users behaviour. Benyon (1995) similarly believed that it is impossible to express or understand user requirements until a fair amount of design work has been done.

Specification of the user interface and human factors requirements presents another difficulty to specifiers. If the specification sets out criteria for usability and learnability that are too stringent, these can prove too difficult and costly to meet (Mantei and Teorey, 1988). Other researchers have found that usability requirements in contracts are vague (Grudin, 1991; Grundry, 1988). In many cases, they are specified by people that do not have an appreciation of human factors and are not specified in a way that designers can take seriously (Stewart, 1991). Grudin (1991) believed that specifications often concentrate on software function and stop short of user involvement. Similarly, Harker (1991) found specifications to be inadequate for the gathering of user feedback.

Perhaps more fundamental is the fact that the specification of the user interface requires definitions of how the software performs in a large number of different low level situations. The difficulty of this task means that the specifier cannot address all aspects of the user interface (Newman, 1991). Lansdale and Ormerod (1994) suggested that coherent specifications are not produced because of the sheer amount of information to be specified and the difficulty of describing this in an unambiguous way. They also suggested that a poor specification can cause problems for the designer, either in terms of too many constraints or not enough guidance.

Another factor affecting specification documents is the level of change inherent in software projects (see section 1.4.2). Tyldesley (1990) reported the need to continuously modify the specification of the user interface. The need to change specifications throughout the software production process adds a further responsibility of communication for the specification writers. When a team member has sifted through a volumous specification once, they will be loath to re-read it each time it is changed and re-issued. The administrative task of keeping specifications up-to-date and team members informed of changes is not a straightforward matter.

1.7 Prototypes

This section begins by outlining the different broad classes of prototype which exist. The current usage of prototypes in software production is then described. The potential which prototypes offer HCI and human factors is then examined. The benefits and problems of using prototypes in software production are then covered.

1.7.1 Classes of Prototype

There are broadly two approaches to using prototypes within software development: revolutionary and evolutionary (Overmyer, 1991). Revolutionary (or throw-away) prototypes present an interactive simulation of the software under production, which is usually iteratively refined until it represents a close likeness of the required software. When the likeness is achieved, the revolutionary prototype is thrown-away and the software is constructed from scratch. With evolutionary prototypes, the software product is constructed by building prototypes with ever increasing functionality which evolve into the end product.

Within this broad definition of prototyping approaches are a great number of alternative definitions of prototyping (e.g., Harker, 1991; Overmyer, 1991). Some of these definitions relate to techniques used to prototype rather than approaches to software production. Harker (1991) classified prototypes as either "static", meaning prototypes which can be demonstrated but not used directly, and "dynamic", meaning prototypes which can be tried out 'hands-on'. These classifications of prototypes are a little misleading because those prototypes classified "static" often demonstrate highly interactive software in an animated way; it is hard to see how such cartoons could be called "static". However, the distinction between prototypes which can be tried 'hands-on' and those which cannot is important.

Wilson and Rosenberg (1988) made similar distinctions to Harker. They suggested that there are three classes of prototyping techniques differing in completeness and testability: storyboard/sketching and slide show techniques; Wizard of Oz techniques; and animated and testable simulations. The classification of non-software techniques such as storyboarding and sketching as prototypes was also considered an appropriate facet of the definition of prototyping by Myhill, Cocker and Brooks (1994).

Key definitions of prototyping therefore describe whether a revolutionary or evolutionary approach is adopted and whether the user will be able to interact 'hands-on' with the prototype or not.

1.7.2 Current Usage of Prototypes

Wilson and Rosenberg (1988) suggested that despite its advantages, prototyping was seldom used. Since then, the introduction of Microsoft's Visual Basic and other 4gls (Fourth Generation Languages), which are well suited to prototyping (Coggman and

Cohen, 1995; Lansdale and Ormerod, 1994), have apparently reversed this trend (Myers and Rosson, 1992). Visual Basic supports both revolutionary and evolutionary prototyping of Microsoft Windows applications. In a survey of 40 projects, Myers and Rosson (1992) found that prototyping was the most common process (46%) for the development of the user interface.

Harker (1991) found that 19 out of 30 projects studied included some form of prototyping, that comprised all in-house projects studied and 56% of bespoke developments. More surprising is Harker's discovery that only half of the prototypes produced were used with end-users.

Overmyer (1991) believed there to be evidence that the evolutionary approach was gaining popularity and felt that this trend was premature. In most cases of rapid prototyping examined, Gordon and Bieman (1994) found that an evolutionary approach was employed. Overmyer (1991) suggested that successful evolutionary prototyping, had been achieved by following a disciplined approach. However, he found that products so built are rated high on ease of use and learning but low on robustness and functionality. Overmyer also found that prototyping produced worse (internal) designs than an approach based on specification. The recently re-introduced RAD method (Stapleton, 1996), which is essentially an evolutionary prototyping approach to software production, has been criticised for lack of robustness and poor design of the final product.

In a large number of cases, the prototyping approach has apparently become a dominant force in the software production. In a survey into the acceptance of prototyping in software development, Kinmond (1995) reported that 52% of respondents were not using a standard methodology and 20% produced no documentation relating to the prototype. Harker (1991) reported that in some cases studied the prototype actually formed the design process and in others, it was ill matched to one. Kinmond's (1995) survey also reported that 37% of prototypes evolved into the final product, 11% were thrown away and the remainder were used for a variety of purposes including re-using code, demonstrations, documentation and training.

Mantei and Teorey (1988) suggested that the advent of prototyping had led to a major upheaval in the traditional lifecycle. Previously, Agresti (1986) had proposed the prototyping approach as an alternative to the waterfall lifecycle. Wilson and Rosenburg (1988), however, suggested that prototyping does not replace the traditional lifecycle methods, rather it complements them. Although this view is perhaps the most balanced and pragmatic, it is apparently not often followed by researchers or in practice. As well as some of the findings already reported, such as developments with no underlying methodology, a further example is given by Mantei and Teorey (1988) who suggested that prototypes can replace diagram designs for the design phase. Unlike others, Wilson and Rosenberg (1988) regard prototyping simply as a tool and not a panacea. Harker (1991) also took a pragmatic view of prototypes, suggesting that a design process needs to have certain characteristics to facilitate prototyping, in particular, it should be iterative to enable changes to be made.

Lansdale and Ormerod (1994) believed that there is some empirical evidence showing advantages of the prototyping approach. Overmyer (1991) similarly found that both evolutionary and revolutionary prototyping can improve software productivity compared with the traditional (waterfall lifecycle) approach to software development.

1.7.3 The HCI and Human Factors Potential of Prototypes

Rudd and Isensee (1994) claimed that prototypes give human factors professionals the opportunity to take the lead in software development for the first time. Iterative prototyping has been described by Coggman and Cohen (1995) as a practical means of achieving improvements in usability, where other proven usability methods could not be applied. The benefits of prototyping are widely accepted as desirable by the human factors community (Harker, 1991).

Overmyer (1991) pointed out that some researchers had been critical of rapid prototyping as in many cases it was used instead of human factors expertise. Examples of this lack in the current use of prototypes are not hard to find. Proponents of RAD do not recognise a human factors element in the evolutionary prototyping approach (Stapleton, 1996). **Only half of the prototypes used in projects studied by Harker (1991) involved users and none of the prototypes were produced by an interface specialist.** Wilson and Rosenberg (1988) also pointed out that programmers and designers are not formally trained in prototyping techniques.

The lack of uptake of the human factors opportunity afforded by prototyping in current practice has several possible causes. Firstly, human factors involvement in software development is still in its early stages and is not widespread (see section 1.8). Secondly, where human factors personnel are involved with a software project, it is usually as specialists acting externally to the software team (see section 1.8.1). Thirdly, human factors specialists often do not have the skills to develop software prototypes (Mantei and Teorey, 1988; Wagner, 1990). Rudd and Isensee (1994) have suggested that human factors specialists should not delegate prototyping to programmers because it breaks the feedback loop: getting an idea, implementing it, showing it to customers, and cycling through again. Mantei and Teorey (1988) also warned of the communication burden incurred when human factors specialists have to work with those producing the prototype. Similarly, Overmyer (1991) suggested that prototyping tools must allow HCI specialists to prototype for themselves.

1.7.4 The Benefits of Prototyping

Recognised benefits of prototyping are described in this section.

1.7.4.1 Requirements Capture and Definition

Two of the main reasons for prototyping is to facilitate requirements capture and improve the requirements definition of a proposed software product. Using prototypes for getting software requirements right is widely reported (e.g., Brooks, 1986; Luff, Heath and Greatbatch, 1994; Overmyer, 1991). Overmyer suggests that revolutionary prototyping is better for identifying and simulating user requirements, because issues surrounding software evolution can be ignored.

The main reason that prototypes can be used to improve the requirements definition for a software product is **that they facilitate communication between software developers and users** (Acosta, Burns, Rzepka and Sidoran, 1994; Harker, 1991; Kinmond, 1995; Wilson and Rosenberg, 1988). In particular, **the prototype provides a common frame of reference for developers and users** (and others) (Miller-Jacobs, 1991; Wilson and Rosenberg, 1988). Prototypes can also be used by developers to help users understand IT proposals (Damodaran, 1991).

The particular value of prototypes is that they place users in a better position to contribute to the software under production (Harker, 1991; Lansdale and Ormerod, 1994) by providing them with a means of understanding the proposed software (Wasserman and Shewmake, 1990).

1.7.4.2 Increased Chance of Software Acceptance

Prototyping is widely accredited with increasing the chances that a software product will be accepted by users and the client (e.g., Mantei and Teorey , 1988; Mason and Carey, 1983). Similarly, Wilson and Rosenberg (1988) believed that prototyping increases the chances that the system will work as expected. Wasserman and Shewmake (1990) also believed that it reduces the likelihood of project failure. This increased chance of software acceptance and success possibly stems from bringing software developers and users closer together and the improvements in the understanding of requirements which this generates.

1.7.4.3 Prototypes as Catalysts for Design Change

Prototypes can prove to be catalysts for design changes even if not shown to users (which studies show many are not). Mantei and Teorey (1988) suggested that prototypes can yield design changes earlier in the development process, with or without user testing. They believe that the number of changes will depend on the complexity of the interface, but that a prototype should always be used on a complex project. This view suggests that constructing a prototype also provides developers with a better understanding of the software they are producing, enabling them to critique and modify the design at an early stage.

1.7.4.4 Flexible Way to Build Software

Some researchers regard prototyping (evolutionary) as a flexible way to build software which copes with vague and changing requirements (e.g., Luff *et al.*, 1994). Both Brooks (1986) and Overmyer (1991) have suggested the use of revolutionary prototypes to establish requirements followed by evolutionary prototyping to develop the software. Although not widely explored in the literature, this approach would seem to represent the best of both revolutionary and evolutionary approaches to prototypes.

1.7.4.5 Reducing Software Development Costs

It has been suggested that prototypes are a means of substantially reducing the cost of a software development (Wilson and Rosenberg, 1988), although such savings are likely to vary according to the type of project. Wilson and Rosenberg believed savings from the use of revolutionary prototyping to be around 3:1. Similarly, Mantei and Teorey (1988) found that cost savings due to making early changes to the software are apparent. They found that fixing things at the prototype stage costs a quarter of what it would when the software has been released. Mantei and Teorey (1988) also pointed out that part of the cost of a prototype should be accounted for as design time, a very valid point which is often neglected. They conclude that prototypes should be constructed when their cost is less than a quarter of the project costs, as design changes later would be likely to cost at least this much.

1.7.5 Problems with Using Prototypes

Recognised problems of prototyping are covered in this section.

1.7.5.1 The Creation of Unrealistic Expectations

One of the main disadvantages of using prototypes in commercial software practice is that they have a tendency to create unrealistic expectations (Kinmond, 1995; Miller-Jacobs, 1991; Overmyer, 1991; Wilson and Rosenberg, 1988). Demonstrating a software prototype in short timescales can lead users, clients and even managers of the software developers, to believe that the software is nearly finished (Miller-Jacobs, 1991). In addition, they may believe that adding new functionality is trivial or that the proposed system will be even better in all respects than the prototype. Adding new functionality to a proposed system may, or may not, be straightforward, but this is not conveyed by a rapidly produced prototype. Furthermore, expectations created by apparent fast response times in a prototype may hide the fact that the proposed software needs to query large databases and will unavoidably have slow response times. Problems arising from the creation of unrealistic expectations can be significant and may include software acceptance failures and dissatisfied users.

Despite the practical problems of changing expectations, Wilson and Rosenberg (1988) were positive about this effect. They regarded the knock-on effect of changing expectations to be changes to product requirements, which they claimed, is a natural effect of prototyping.

1.7.5.2 The Difficulty of Producing 'Hands-On' Prototypes

Producing a prototype which is robust enough to be tried out by users 'hands-on' is not trivial. It is also very time consuming and therefore expensive. Wagner (1990) found such prototypes to be the most complex to create. Although Wagner was able to use Apple HyperCard for interactive prototyping, it was found to be insufficiently robust for substantial user testing. Wilson and Rosenberg (1988) concluded that such prototypes are rarely produced, because they are time consuming and are superfluous. However, the type of feedback which is gained from a user when showing them a prototype to illustrate the software concept, is different from the detailed usability testing which could be done with a 'hands-on' prototype

1.7.5.3 Constraints and Limitations

Constraints and limitations that apply to the real software product can be ignored when constructing prototypes (Glushko, 1992; Wilson and Rosenberg, 1988), which can obviously led to the creation of unrealistic expectations. Conversely, the real software product can suffer because of constraints and limitations that apply to the prototype development language (Glushko, 1992; Overmyer, 1991).

Luff *et al.* (1994) put forward the view that evolutionary prototyping still relies on some formulation of the scope and functionality of the final system (even if it is preliminary) and that this shapes subsequent iterations in the design.

1.7.5.4 Controlling and Managing a Prototyping Process

Wilson and Rosenberg (1988) suggested that a prototyping process may be difficult to control and manage. Overmyer (1991) agreed, believing this to be due in part to the high visibility of prototyping. Overmyer also suggested that a particular problem of revolutionary prototyping is knowing when you have a complete enough model of the proposed software, making it hard to avoid over-engineering. Controlling the number of iterations of the prototype that are carried out was also a concern of Lansdale and Ormerod (1994). Mantei and Teorey (1988) also highlighted the need for carefully controlling the process as the prototype can lead the designer to add "bells and whistles".

1.7.5.5 Integration with Software Engineering

Prototypes usually exhibit disparity between stated methodology and engineering practice (Overmyer, 1991). Miller-Jacobs (1991) also described problems caused by prototypes infringing on the design process that is supposed to follow specification. The practice of using prototypes is further at odds with the context of software development, because the current form of contracts cannot deal with prototypes (Miller-Jacobs, 1991). However, in some situations a more flexible approach to contracts and evolutionary software production has proved to be possible (Stapleton, 1996).

1.7.5.6 Dealing with Changes Deemed Necessary from Using Prototypes

Using prototypes to shape a software product will generate changes to the software under production. As Grudin (1993) pointed out, prototyping has little point if design cannot be changed. However, reluctance to make changes can be apparent if prototypes are run late (Harker, 1991). Lansdale and Ormerod (1994) concurred that late design change is a problem. Overmyer (1991) was also concerned that it is the nature of evolutionary prototyping to freeze the design too early which leads to developers' reluctance to make changes.

1.7.6 Summary and Final Commentary

Both evolutionary and revolutionary prototypes are used in commercial software development. Other classifications of prototype also exist, the most important of these is whether or not the user can interact with the prototype 'hands-on'.

The introduction of Microsoft Visual Basic and other 4GLs, which allow rapid and accessible development of Windows applications, has boosted the use of prototyping, both evolutionary (e.g. the re-emergence of RAD) and revolutionary.

A study published in 1991 found 19 out of 30 projects analysed to involve prototyping, but only half of these had user involvement. A survey published in 1992 found that prototyping was the most common process for developing user interfaces. Clearly, prototyping is in widespread use in software production. However, in many cases the prototype is the development process, and in others, the technique is ill-matched to one.

Evolutionary prototyping is often criticised, because it is seen as replacing the need for HCI or human factors involvement in a software project. Furthermore, software products so constructed are rated high on ease of use and learning but low on robustness and internal design.

Prototyping is believed to be a means of HCI and human factors specialists taking a more dominant role in software production. However, the support role such personnel often perform does not usually allow them to realise the full potential of the technique. The benefits of the technique are widely recognised by the human factors community.

There are many widely recognised benefits of prototyping. One of the most important is that prototyping has probably become the best way of generating, capturing and understanding requirements for a software product. The main reason for this is that prototypes provide a common representational currency for developers and users. The use of prototypes in requirements analysis is strongly believed to improve the chances of software acceptance. Whether or not utilised with users, prototypes generate design changes earlier in the production process. This is likely to be because by making abstract software concepts concrete, prototypes help members of the software team to visualise and gain a deeper understanding of the software under production than they would get otherwise. Prototypes are also seen as a means of reducing development costs, primarily by getting requirements right and discovering necessary design changes early. A final benefit of evolutionary prototyping is that this is a flexible way to build software which copes well with the nature of software production.

The disadvantages of prototyping identified in the literature are of considerably less significance than the reported advantages. However, some disadvantages are serious and these include the creation of unrealistic expectations. Rapidly prototyping complex software can lead to people (user, clients, software managers, etc.) believing that the software is nearly finished, that modifications are trivial and that software response times will be fast. The illusion presented by some software mock-ups can create expectations which are not achievable and can be damaging if not managed extremely carefully. In order to properly evaluate a user interface, it is necessary for the users to operate it 'hands-on'. Constructing such prototypes requires a significantly greater investment than prototypes constructed to demonstrate concepts. It is therefore believed that such prototypes are rare. A further problem with prototyping is that constraints and limitations of the prototype language may influence the design, or constraints and limitations of the implementation language may be ignored by the prototype. The management of a prototyping process is considered to be difficult — another problem with the technique. Furthermore, prototyping does not fit well within software engineering processes, such as those imposed by the underlying waterfall lifecycle, or the prevalent contract and specification centred software procurement. Finally, dealing with the software changes, which the use of prototyping has highlighted, can present problems within a traditional software process, or even in an evolutionary development when time is running out.

Prototyping is thought to be a major advance in software development, primarily because it helps to get the software requirements established correctly by providing users, clients and developers with a means to communicate and understand each other. Some disadvantages of prototyping are identified and these must be attended to, but ultimately, these are believed to be heavily outweighed by the advantages.

1.8 Current Commercial HCI Practice

This section examines which HCI techniques are currently in use and the difficulties encountered implementing these techniques (cf. section 1.9 which describes HCI theory and its level of application to current HCI practice).

Branscomb (1983) stated that although it was realised that the fundamental architecture[2] of software has a profound influence on user friendliness, next to nothing was known about how to make architectural decisions differently, in the interest of good human factors. Current commercial HCI practice is only now beginning to find ways of influencing such architectural decisions. Karat (1996) has recently claimed that more companies are subscribing to the concept of User-Centred Design, making it more explicit and integrating it within the development process. Norman (1996) has also recently expressed the belief that more companies are embracing "what we do", even though they may not understand it yet. Thus, some companies are apparently beginning to take HCI and human factors more seriously. Exactly what this means will be explored below.

1.8.1 The Mode and Context of HCI Practice

HCI designers and human factors personnel are usually employed as specialists, not as part of the core software production team, acting in a support role (see section 1.10.1). This support role dictates the kind of involvement that such specialists usually play in a software team and the context of that involvement. Whilst there are signs that HCI designers are beginning to be incorporated into core production teams, this is rare and references to such roles are at best only hinted at in the US literature. Some researchers even believe that there would not be enough human factors people to go around if this approach was taken (e.g., Eason and Harker, 1991).

Karat (1996) believed that skills in applying HCI within this practical context are currently more "in the heads of practitioners" than they are in the HCI literature.

1.8.2 Perceived Lack of Value of HCI

Many researchers have cited a lack of uptake of HCI in industry (e.g., Browne, 1994). Craig (1991) suggested that the idea that design is not an afterthought took a long time to establish. This would imply that design is now universally accepted as not being an afterthought, but this is not the case. As Grudin (1993) pointed out, there persists a belief that the interface can be ignored or tidied up at the end, a fact which causes practitioners to often complain about late involvement in the production process. There remains a widespread lack of understanding of HCI, human factors and the need for

[2] In this context 'architecture' is taken as meaning the internal structures of the software, Brooks (1975) uses the word 'architecture' to mean the user interface.

user involvement. Stewart (1991) agreed that there is still a misunderstanding of what human factors is and why it is important. Insufficient appreciation of the contribution of human factors engineering (and its benefits) is cited by Hakiel (1995) as just one inhibitor to a more widespread uptake of human factors and an integrated approach to product engineering. Hakiel suggested that in practice, usability is considered to be a testing activity rather than a product design activity. This is indicative of the mode and context of HCI practice and the usually external (or 'outsider') positioning of the HCI designer or human factors specialist, which is in agreement with the usual situation described by Carey *et al.* (1991).

Carey *et al.* (1991) believed that demand for HCI expertise from an in-house central resource can soon exceed supply. This level of demand is not apparent across the industry, where, in comparison to programming jobs, HCI design jobs are almost impossible to find. The job market is perhaps the clearest indication there is, of a lack of perceived value and uptake of HCI and human factors in industry.

1.8.3 User Interface Design in Practice

Browne (1994) suggested that, as has historically been the case, user interface design is often left to the whims of various parties guided by project deadlines and ease of implementation. He further suggested that even where HCI techniques are used they are not used as intended. For example, prototypes are created but not shown to users. There is some evidence that Browne is right. Advocates of RAD (e.g., Stapleton 1996) talk of the need for user involvement and iterative design, but treat HCI and human factors as completely foreign.

Gould and Lewis (1987) claim that some designers intend to follow guidelines and focus on the user but do not, or they think they are, when they are not. It is apparent that the 'designers' referred to by Gould and Lewis are IT designers, more akin to programmers than HCI designers. A further finding of Gould's (1988) work was the discovery that principles of behavioural design (which he considered trivially simple to follow) were not common sense to designers and were difficult for managers to deal with.

The work of Gould and Lewis (1987) and Gould (1988) makes it clear that the user interface designers they studied were actually IT designers charged with some user interface responsibility. Hence the reason why behavioural design principles were so foreign to them. It is likely that such designers (untrained in interaction design) make generalisations about users and tasks. Stewart (1991) described such designers as having naive views of users and tasks and their overuse of the words 'user' and 'task'. Shackel (1991) suggested that the term 'end users' betrays an attitude (viewing the user as a 'peripheral' of the system) which causes bad design and usability failures. Norman (reported in Rheingold, 1990) also believed that software design tends to be done by people who have, "off-the-top-of-their-heads ideas and beliefs about imaginary beasts called 'the users'" (p. 5).

Some researchers and practitioners believe that there is currently little HCI practice (Earthy, 1992). Cockton (1991) believed that HCI design and evaluation are not widely used within structured development. This may be partly due to the fact that the linear sequence of events in the waterfall lifecycle precludes effective evaluation. Grudin (1991) also found that HCI practice is particularly lacking in software produced under contract (such software usually being produced by a waterfall type lifecycle) where there are formidable barriers to user involvement.

Coggman and Cohen (1995) found that time and resource constraints limit human factors involvement and prevent proven usability techniques being applied in practice. It is the level of difficulties in the practical application of HCI that led Nielsen (1993) to develop 'discount usability' techniques. These provide a means of prioritising and optimising HCI involvement on a software project, to cope with severely limited time and resources. An example of such a technique is heuristic evaluation of the user interface by HCI specialists. The practicality of this technique means that it probably has a stronger foothold in current HCI practice than is apparent from the literature. Levi and Conrad (1996) reported adoption of the technique, because it enabled a quick and cheap evaluation. However, this is founded on their belief that the skills required to carry out an heuristic evaluation could be taught in a day, which is a gross over-simplification. They cited research which found that 40-60% of usability problems which would be found using an empirical approach can be found using a heuristic evaluation. This would be a surprising result if heuristic evaluators had only received one day's training. Nielsen (1992) had previously acknowledged the fact that some heuristic evaluators are better than others, with his definition of a "double-specialist" as a user interface expert with experience of the application domain. Nielsen found that two such specialists should be able to identify 80% of the usability problems which exist in an interface.

A further description of the activities of today's HCI designers can be found in section 1.10.1.

1.8.4 The integration of HCI into Software Production

Bell and Spencer (1995) believed that current software engineering methodologies fail to meet user requirements and support the design of GUIs. The revived RAD approach to software production addresses both of these issues without reference to HCI or human factors. Sutcliff (1992) suggested that HCI is not seen as part of systems engineering, while Hakiel (1995) believed that an integrated approach to the design of human-computer systems, explicitly recognised by the discipline of human factors engineering, is not yet routine in the software industry.

Some researchers in HCI and human factors have begun to package their work in a manner which they believe will be more acceptable to the software engineering community (e.g. Lim and Long, 1994a; Lim and Long, 1994b). The ultimate aim of such efforts is to integrate HCI and human factors with software production. The MUSE method, described by Bell and Spencer (1995), supports the integration of human factors and software engineering activities by specifying "handshaking" (i.e. cross-checking) between the two activities. However, this form of integration both acknowledges and perpetuates the role of the HCI or human factors specialists as separate from the main production process. This leaves open the possibility for the user to still be considered as a peripheral to the system rather than the centre of it.

Some researchers acknowledge the fact that, in practice, an element of iteration is inevitable within the linear waterfall lifecycle. Lansdale and Ormerod (1994) believed that to accommodate some of the difficulties associated with developing software using the waterfall lifecycle (see section 1.3.1), it is common that a project will involve some degree of iteration between stages. Myhill, Cocker and Brooks (1994) also acknowledged the constraints imposed on commercial software development practice by the waterfall lifecycle and proposed a software production process which encouraged iteration (and the use of prototypes) within each of the waterfall phases. The packaging of human factors efforts as deliverables within the software production process is proposed by Hakiel (1995), partly because such an approach would be viable regardless of the software production process adopted.

1.8.5 Lack of Resources for Practical HCI

Resources to support HCI in practice are rare. Baeker and Buxton (1987a) claimed that there is a lack of appropriate textbooks. This is further illustrated by the fact that journal articles are not written in a tutorial or procedural way Gould (1988). This issue is taken up in more detail in the section on HCI theory (see section 1.9).

1.8.6 Summary and Final Commentary

Evidence suggests that companies are beginning to take HCI and human factors more seriously. However, HCI and human factors specialists usually function in a support role and are rarely employed as part of the software team. Evidence also exists to suggest that industry still does not understand the value of HCI and human factors. This can partly seen by the lack of jobs advertised in these fields in the software press compared with the vast number of programming jobs. It appears that a large proportion of interface design work is carried out by software engineers. In order to gain acceptance into software production, some researchers have developed HCI and human factors techniques which are specifically designed to fit into the commercial software production context that exits. Finally, resources available to the HCI practitioner are scarce.

1.9 HCI Theory

Researchers tend to agree that HCI theory is rarely used in practice. Some believe one reason for this to be that some of it has no application in commercial software development practice. Others consider the reason to be that HCI theory is not accessible to HCI practitioners and not taught to students. Many researchers agree that HCI research has shown that a gap exists between HCI theory and practice.

1.9.1 HCI Theory is Not Applicable in Practice

Bannon and Bødker (1991) summarised their view of the HCI research contribution to date. As they saw it, the contribution criticised current design practice for not paying enough attention to users, offered general and not very usable guidelines and speculated on alternative ways of doing things without much practical grounding. This view suggests that HCI theory was not considered useful in practice. Carroll, Thomas and Malhotra (1980) had previously found that solving unstructured problems (like design) could not be explained with existing theory.

Researchers have also considered the application of psychology theories to aspects of software production. Sheil (1987) found psychology theory to “lack the robustness and precision” necessary to predict behaviour as complex as programming. More recently, Lansdale and Ormerod (1994) suggested that it is the context sensitivity of user interface design which makes the application of psychological theories very difficult.

Some researchers argue that there is no ‘theory of HCI’ (Dowell and Long, 1989), as they believe that HCI practice is a craft discipline (see section 1.10.1.1 for further discussion of this).

1.9.2 HCI Theory is Not Applied in Practice

It is also apparent from the literature that some researchers have concentrated more on the fact that HCI theory has not been applied in practice. Bellotti’s (1988) survey into the use of HCI methods in design showed that, although designers (mostly non-HCI specialists) recognised the need to consider the user, there was little evidence that they used any modelling, analysis or evaluation approaches from HCI theory and research. Firstly, she suggested two reasons why theoretical approaches are not being taken up by designers (1993); secondly, she claimed that such theory is too narrow in scope and that it is unclear how theories relate to one another or to design problem solving. Bellotti (1993) described a credible new “semi-formal notation” for recording design rationale, which attempts to bridge a gap across the various disciplines involved in software production. However, her ultimate conclusion was that further work remains to be done in improving the process of bridging.

Green, Davies and Gilmore (1996) described the failure of cognitive psychology to make significant contributions to the study of HCI in general or contribute greatly to the design of interactive artifacts.

1.9.3 HCI Theory is Not Accessible in Practice

Brooks (1990) believed that research papers are not written in a way which is accessible to practitioners, partly because the papers tend to present an innovation in as abstract and general form as possible. **Brooks strongly advocated the use of case studies describing the organisational context as well as research as a means of propagating innovation among practitioners**. It is precisely because the application of theory is context dependent that Bødker *et al.* (1993) recommended the use of example driven presentations as more appropriate than stating general guidelines and methods.

Karat (1996) believed the contribution of human factors people to producing usable software to have come a long way. However, he believed that exactly what this contribution consists of is currently more "in the heads of practitioners" than in the HCI literature or academic training. This is perhaps further support for the fact that the skills of human factors people and HCI designers have a craft-based element to them (see section 1.10.1.1).

1.9.4 The HCI Theory-Practice Gap

One development in HCI research observed by Bannon and Bødker (1991) was the search for theoretical frameworks and the subsequent realisation of the existence of a gap between theory and actual use situations. Mantei and Teorey (1988) aimed to, "fill the current gap that exists between the human-computer interaction research papers and the pragmatic needs of the software developer." (p. 438). Their paper aimed to present systems analysts and project managers within the costs and benefits of applying human factors to software development.

Two of the three main objectives of work by MacLean, Young, Bellotti and Moran (1991) were the bridging of gaps. One gap was that between HCI theory and the practicalities of designing software artifacts. The second was a conceptual gap between behavioural and computing disciplines. Buckingham-Shum and Hammond (1994) suggested that the HCI theory-practice gap (which they refer to as a 'gulf') comes from the fundamental difference between science and design. They expressed the view that HCI researchers and practitioners have different goals and different languages, which causes them to make use of different conceptual tools.

Human factors researchers have been accused of not considering and not understanding their users, the designers. Stewart (1991) reported that designers' complaints about human factors literature included: it is incomprehensible; it is pseudo-scientific and full or jargon; it is difficult to apply to their own problems; and is difficult to find in the first place. HCI and human factors researchers have been criticised for not appreciating the realities of HCI practice. For example, methods which have been used by outstanding researchers to improve designs have not fared so well in trials with less brilliant users (Fischer, Grudin, Lemke, McCall, Ostwald, Reeves, and Shipman, 1992). Such criticisms could be considered harsh in the current context of HCI and human factors involvement in software practice. Often interface design is not carried out by HCI or human factors specialists (see section 1.8.3 for a description of current user interface design practice) so the designers criticising human factors and HCI theory could well have no prior experience of these disciplines. Even when HCI or human factors specialists do have a role in software practice, it is usually so limited (see section 1.10.1 for a description of the usual role of HCI designers and human factors specialists) that making use of even the most basic theories is difficult. It is therefore likely that the theory-practice gap exists for a number of reasons, one of which may be theorists' lack of appreciation of commercial practice, and another is likely to be the current limited context of commercial HCI practice.

The image of HCI theorists purporting ways to improve the communicability of software may also damage the perceived credibility of their work. As in many fields, HCI academics are diverse, and some are not adverse to ignoring basic design guidelines, e.g. using inappropriately designed presentation material. In a communication discipline, practitioners may not readily accept such obvious oversights from those claiming to be authorities in the field.

1.9.5 Summary and Final Commentary

Some researchers believe that HCI theories have no applicability to the practice of software design. Others even suggest that there is no theory of HCI because it is craft based. The majority of researchers believe in the existence and applicability of HCI theory but realise that it is not being applied very much in practice. Some think that one reason for this lack of application is the inaccessibility of HCI theory to practitioners. Concern has been expressed that a gap exists between HCI theory and practice.

1.10 People in Software Production

For a considerable number of years, many researchers have believed that the key to producing good software lies with the people in the software production team. Development processes, techniques and methods are all of considerably less importance than the people in the team. As the title from Bach's (1995) paper illustrates, "Enough About Process: What We Need Are Heroes".

This section covers HCI designers and human factors personnel and includes a pragmatic view of their role. Computer programmers (including IT designers) are discussed as well as the evident individual differences which exist among them. Aspects of software teams are also examined. Finally, calls for a new type of specialist to work in software teams are identified.

1.10.1 HCI Designers (and Human Factors Engineers)

There is considerable evidence in the literature to suggest that HCI designers and human factors engineers usually function as specialists and advisors *outside* the software production team (examples include, Carey *et al.*, 1991; Coggman and Cohen, 1995; Lansdale and Ormerod, 1994; Stewart, 1991). Further evidence indicates that it is common for these advisors to function from central support groups (Bannon and Bødker, 1991; Carey *et al.*, 1991; Eason and Harker, 1991; Grønbæk *et al.*, 1993). Bannon and Bødker (1991) described this traditional role as limited; others found that HCI designers are often brought in too late (Grønbæk *et al.*, 1993; Landsdale and Ormerod, 1994); Grudin (1993) claimed that user interface specialists rarely have the big picture. Given their position as outsiders to the team and their late involvement in the software production, this is hardly surprising. As Lansdale and Ormerod (1994) suggested, what is needed is their input at the concept stage.

The work of some researchers implies that they consider an HCI or human factors role to be part of the software team. Erickson (1990) described a team member with responsibility for "taking the user's point of view". Similarly, the 'architect' (a description effectively meaning a user interface designer) defined by Brooks (1975), was said to be the users' agent representing their interests in the inevitable trade-offs that occur in implementation. Brooks believed this role to be a full-time job. The literature contains few references to the involvement of HCI or human factors personnel as full-time team members.

Many researchers have found that the volume of enquiries to an internal HCI or human factors consultancy group rapidly outstrips the available resources of the group (Carey *et al.*, 1991; Eason and Harker, 1991; Stewart, 1991). This finding may suggest that the value of HCI and human factors involvement becomes clear when a group is established, but companies without such departments are ignorant about what they are missing out on (as illustrated by the comparative lack of such jobs in the software industry). One solution to this problem would be to assign an individual with an HCI or human factors role to the software team for the duration of the software production, from initial requirements analysis though to installation and training.

The nature of the usual HCI involvement leads some researchers to believe that the role of an HCI designer is almost complete when the specifications are written (Browne, 1994). This view is largely unrealistic and probably stems from the 'outsider' situation of most HCI designers. The reality of the situation is that no product is ever predictable and there are always issues concerning usability that arise only during implementation (Brooks, 1975; Tyldesley, 1990). Therefore, involvement of the HCI designer is required throughout the implementation process to ensure that usability issues are resolved in the interests of the user rather than the technology.

Perhaps the most forward looking HCI role has been established by Norman (1996) at Apple Computer. Norman described the group where he works as the "User Interface Architects Office" and its role, as making high-level structure and functionality considerations. However, the function of this new group still appears to be external to the core software production team.

Brooks (1995) believed that the architect (user interface designer) is like the manager and director of a motion picture and suggested that having a system architect (user interface designer / architect) is the most important single step toward achieving conceptual integrity. Nelson (1990) also viewed the skills required for user interface design to be like those required for movie production, i.e. not technical but rather, to do with conceptualisation. The movie analogy to software design is not uncommon (see also Heckel, 1991) and extending it to describe the current role of interface designers may produce the following analogy. If a movie producer (Brooks' term, 'director' may be better) performed a role analogous to the majority of modern day user interface designers, their role on the production of a film would be over when the script is complete. They would perhaps get an invite to the preview, but by then, it would be a little late to make any changes.

1.10.1.1 A Pragmatic View of the HCI Role

One practicality of the HCI role is that commonly, HCI or human factors people have no authority to decide on the inclusion or exclusion of functionality (e.g., Coggman and Cohen, 1995). Furthermore, Stewart (1991) suggested that because much design is *ad hoc* and decisions are made fast, the reality of the situation is that the human factors specialist (acting externally to the team) may not be around when the decisions are made and subsequently it may be too late to change them. Stewart also suggested that sometimes the team may wait until the human factors specialist is not around to make some key decisions; in the competing priorities that exist, it is hard to put usability over functionality.

One reality of the current context of HCI design in practice is that it is often not done by HCI specialists. Harker (1991) analysed 30 software projects and found that 19 did some form of user prototyping. However, she also discovered that none of the prototypes were produced by interface specialists.

Mantei and Teorey (1988) pointed out that human factors people do not necessarily understand software very well. Similar problems involving graphic design specialists to the development process were found by Grønbæk *et al.* (1993). A full year was required to integrate them into the team. Tyldesley (1990), coming from a User Interface Group at Digital, believed that user interface personnel should be prepared to do some code level work themselves and must attend design team meetings. Clearly, he believes that human factors people should increase their effectiveness by gaining technical skills.

Bannon and Bødker (1991) described the limited scope of current HCI involvement coming from centrally placed human factors personnel, comprising task analysis and possibly later, display and layout considerations.

Coggman and Cohen (1995) expressed the view that the human factors role needs support from management. The need for such support is clearly essential to enable the human factors voice to be heard.

Brooks (1975) expressed concern about separating the responsibility for writing the specification (i.e. by the architect, meaning user interface designer) from building a fast cheap product. In such situations, he asked, "what discipline bounds the architect's inventive enthusiasm?". The separation of HCI and human factors from the core software team effort undoubtedly incurs such difficulties.

Many researchers believe that modern HCI practice and software design involves an element of craft skill (Baeker and Buxton, 1987a; Dowell and Long, 1989; Heckel, 1991; Rubenstein and Hersh, 1987; Wroblewski, 1991) and others cite programming as a craft (e.g., Brooks, 1975). Wilson and Rosenberg (1988) suggested that in the design of a user interface the rapid prototyping is the equivalent of the sculptor's clay. If the HCI designer can be regarded as a craftsperson, it is one of the few crafts where the craftsperson must communicate the workpiece to others who would construct it (perhaps the architect of a building is similar). The hands-off (the actual implementation) nature of the HCI designer's craft is revealing of the frustrations which are ever present in the role.

'Intuition' is not irrelevant in current user interface design practices. Historically, because computer users were very like software developers, their intuitions were good (Grudin, 1993). Although such intuition is less relevant today, developers still base their understanding of users on it (Grudin, 1993). However, intuition is not to be too easily dismissed and is likely to be more prevalent in user interface design than is readily admitted. The design of the Apple Mac involved various intuitive design decisions (Levy, 1995). Gould and Boies (1987) report that they had an "intuitive belief" that designing a principle support office system was possible.

1.10.2 Programmers

This section is entitled ‘programmers’ but is intended to encompass roles described in the literature as software engineer, software developer, programmer and IT designer (such designers are often senior or experienced programmers).

Although huge individual differences among commercial programmers are apparent (see section 1.10.2.1), generalisations about programmers in the literature and research are the norm. Whilst such generalisations are of questionable validity, a selection of these does provide an illustration of commercial programmers:

- “Developers tend to be young, rationalistic, idealistic, and the products of relatively homogeneous academic environments.” (Grudin, 1993, p. 107);

- programmers were found to have higher needs for personal growth and development than any other job category previously measured, and had lower needs for social interaction than those in most other jobs (Couger and Zawacki, 1980);

- until their program works, programmers feel obsessed, they are unlikely to go home when their program has just crashed (Kim, 1990);

- a programmer will not trust you unless you have written a program and experienced the basic drama of so doing (Kim, 1990);

- division of labour separates programmers from the outside world (Gabriel, 1994; Grudin, 1993);

- the status of a programmer in a team is usually strongly influenced by their *abilities as perceived by others* (Weinberg, 1971), which can lead to defensive behaviour and blame attribution activities;

- developers are sometimes uncomfortable about criticisms (Levi and Conrad, 1996) and are often *proud* of their user interface designs, regarding the adoption of standards as a *threat* to both their creativity and potential monetary rewards (Billingsley, 1988);

- some developers do not see the user interface as their responsibility and prefer to concentrate on gaining more marketable technical skills (Perlman, 1988);

- “programmers are notoriously bad at estimating time scales” (Stephens, 1996) supporting Brooks’ (1975) view that programmers estimate based on the type of programs that they program and run themselves (this view is partly misleading as programming involves making estimates within the context of the existence of many unknowns);

- programmers are unlikely to be good at interface design as they are too influenced by their underlying knowledge of engineering models (Browne, 1994); programmers tend to think more about internal structure, simplicity of internal design and therefore become functionality focused (Heckel, 1991); the concept of what 'works' in the user interface has a purely logical emphasis to a programmer (Heckel, 1991);

- programmers have been known to code to an incorrect specification to achieve compliance with it (Grudin, 1991);

- software engineers may lack empathy or sympathy for inexperienced or non-technical computer users (Grudin, 1993);

- the different values, work styles and even languages of developers and users can hinder communication between them (Grudin, 1993);

- "...most software designers are both poor writers and poor designers of friendly software. I don't think this is a coincidence: both are forms of communication. The average software designer has the logical skills of a software engineer but not the visual thinking skills of a communicator." (Heckel, 1991, p. 127).

The views of some researchers and practitioners in HCI towards the technical core of people that actually write software clearly illustrates the bad attitude and lack of respect which can dominate the interaction between these disciplines:

"*Historical accident has kept programmers in control of a field in which most of them have no aptitude: the artistic integration of the mechanisms they work with. It is nice that engineers have found a new form of creativity in which to find a sense of personal fulfilment. It is just unfortunate that they have to inflict the results on users. Learning to program has no more to do with designing interactive software than learning to touch-type has to do with writing poetry... ...what we need in software is what people are taught in film school.*" (Nelson, 1990, p. 243)

Norman (1990) counselled that we should, "*Keep the technologists busy... But make them keep their hands off product design.*" (p. 217)

This section has tried to provide an illustration of commercial computer programmers. Many of the statements made about programmers by the research community involve generalisations, which are not in keeping with the findings in individual differences research. However, most practitioners working in software would probably recognise many of the generalisations cited as well as the attitude and frustrations which drives comments like those from Nelson and Norman. It should also be noted that the dominance of GUIs has made the programming task more difficult (Myers and Rosson, 1992).

1.10.2.1 Individual Differences

In a cost estimating study of a commercial software development spanning several years, Boehm (1981) found that differences in personnel and team capability were the most significant factors affecting programming productivity. Gabriel (1994) highlighted this productivity differential by describing the Borland software team that produced Quattro Pro for Windows (QPW). Gabriel explains that the QPW programmers were very highly skilled and each produced around 1000 lines of production code a week. He claimed that the average US programmer produces 1000-2000 lines of production code a year. Although measuring programmer productivity in lines of code is a very coarse measure (a good programmer can often write more compact code), it at least gives an indication that potentially large differences in productivity do exist.

Curtis has been involved in a considerable amount of research associated with programmer performance (examples include, Curtis, 1981; Curtis, 1986; Curtis *et al.* 1987; Curtis, 1988). His early work described a study where performance differences between programmers over various tasks was found to be of the order of 22:1 (Curtis, 1981). Removing outliers, Curtis found that differences of 13:1 remained. Thus, Curtis believed he had substantiated the fact that large performance differences were evident among commercial programmers as had been suggested by Sackman (1970). McGarry (1982) also found differences of the order of 23:1 among NASA programmers. Weinberg (1971) also believed individual differences in strengths and weaknesses of programmers to exist. Similarly, Curtis (1988) concluded that it is "strikingly apparent" that programmers are different from each other in large ways. In one study he found that half of the variance in comprehension performance came from individual differences. Sheil (1987) similarly reported to being particularly struck by the existence of participants whose scores on some measures were far outside the range for the group they belonged to (outliers).

Although it is believed that these individual differences come from motivation, experience, intelligence, etc. (Curtis, 1981), the individual differences paradigm has failed to show why the differences exist or how to reduce them other than through selection (Curtis, 1988).

A further dimension of the individual differences found includes "tremendous variability" in how tasks are carried out by programmers (Curtis *et al.*, 1987). This is a factor often lacking in the analysis of differences between programmers' performance. The programming role is rarely just about purely technical performance of the computer programming task. Brooks (1975) realised that a lot of problems experienced in software development had to do with communication. Looking at the whole of the programmer's role does not usually form part of individual difference studies. There is clearly far more scope for variability among programmers than is apparent from many studies. For example, informal roles which programmers play in a team because of these differences have been highlighted (Myhill, 1993).

One consequence of the individual differences result and the finding of outliers is that some computer programmers are superb compared with their contemporaries. There is evidence in the literature to support this supposition. Curtis (1988) suggests that the success of the 'Chief Programmer Team' (see section 1.10.3) depends on the availability of a "superb technician" to be the 'Chief Programmer'. The 'Super-Conceptualiser' is a term introduced by Curtis *et al.* (1987) to describe exceptional individuals in some software teams with rare expertise allowing them become the "keeper of the project vision". However, such strong conceptualisation ability was apparently not linked with strong programming ability as Curtis *et al.* reported that super-conceptualisers admitted that they were not good programmers themselves. Brooks (1986) highlighted the need to identify the "great conceptual designers of the rising generation", such designers probably being senior programmers (or at least very experienced programmers). Brooks (1995) reaffirmed his belief that **the most important action of a software development is "commissioning one mind to be the product's architect, who is responsible for the conceptual integrity of all aspects of the product perceivable by the user**." (p. 256).

The existence of outliers at the lower end of the performance scales shows that some commercial programmers are very poor performers compared with their contemporaries. Even without outliers, the variability of 13:1 reported by Curtis (1981) indicates a wide distribution of ability among computer programmers. The fact that selection of programmers is extremely difficult is apparent from the existence of such differences among commercial programmers. The consequence of the existence of individual differences among commercial programmers is rarely alluded to in the literature. The reality that software production must contend with is, therefore, **mixed ability teams.**

1.10.3 Software Teams

Curtis (1988) suggested that there had been too little research on software teams compared with their impact on software productivity. Given the importance of software teams this probably still holds true. An organisational obstacle to User-Centred Design reported by Karat (1996) is being able to create and maintain teams of people with the range of expertise necessary to solve complex design problems. Curtis *et al.* (1987) believed that crucial aspects to the success of a software team are personnel selection, assignment, education and communication. However, in practice, personnel selection is clearly difficult as is evident by the huge differences in performance of commercial programmers which exist (see section 1.10.2.1) and assignment is often driven by availability of resources - hand-picked teams are rare.

Many of the most significant software products produced in the history of software have been produced by individuals or very small teams (two or three people). Such software includes, the C programming language, Zortech C++, Lotus 123, and dBase (Thielen, 1991). Although the complexities of programming GUIs and Windows software make this apparently less feasible, examples of big feats from small teams are still in evidence. For example, Quattro Pro for Windows (QPW) was developed by a core team of four programmers (Gabriel, 1994). It is unclear whether the average size of a software team today is any different than it was 20 years ago. Karat and Bennett (1991) suggested that most software development involves a fairly large group of people.

Following on from this, it has been suggested by Whiteside *et al.* (1988) that commercial software developed then, with complex development processes, could involve hundreds or even thousands of people. Stephens (1996) described the changing nature of software development from small numbers of “hackers in backrooms” to teams of 20-50. However, Brooks (1975) described the development of the OS/360 operating system software which at its peak had over 1000 people working on it, and went on to propose that 10 people would be a sensible maximum size for a software team. In support of this, Scott and Simmons (1975) found that team productivity levelled off at 9-12 people. Throughout the history of software, there have usually existed examples of software projects with both small and large numbers of people working on them. The comments of Karat and Bennett are perhaps the most accurate, because in software terms, 10 people is a lot to co-ordinate and to collaborate effectively. Harker (1991) believed that the major advantage of small development teams is ease of communication and reduced potential for ambiguity and misunderstanding in relation to the specification. In order to extend the benefits of small teams to slightly larger teams, two key team structures have evolved.

The ‘Chief Programmer Team’ is a particular team structure, which places central responsibility for programming and technical communication with a chief programmer, with the rest of the team members organised as a support team (Baker, 1972; Mills, 1971). Brooks (1975) used an operating theatre analogy and referred to such a team as a ‘Surgical Team’. Another team structure which arose is the ‘Egoless Team’ in which responsibility for the software is shared among the team members such that no individual feels private ownership of any particular piece of the program (Weinberg, 1971). These team structures were found to be suited to different types of software development situations, a summary of which may be found in Curtis (1988). Both team structures were also believed to be most efficient operating with 12 or less people. There is evidence that Microsoft currently adopts an approach to software development based on the egoless team. McCarthy (1995) (a director of Microsoft’s C++ Product Unit) advocated design such that nobody owns the specification, as well as a team mix of six developers, three quality assurers, one program manager and two writers. Wider issues surrounding the organisation of software development teams also appear to be relatively unchanged.

Curtis *et al.* (1987) found that it is common for the early stages of a software project to be dominated by a few individuals but found that this reliance on a few people is less troublesome in practice than would be expected. In certain situations, Grudin (1991) noted, it is common for the design and development to be carried out by completely different teams, once the project definition is established one team moves on and another begins its work. This approach clearly creates considerable communication difficulties during the life of the project. The kinds of difficulties this creates are similar to those when individual team members change during the design or development phases. Harker (1991) described the effect this can cause as “design drift” as the new team members introduce new ideas and assumptions. Similarly, Grudin (1996) suggested that new arrivals in a team can be a problem as they challenge the existing design and introduce new ideas. He further pointed out that such new arrivals also waste team members’ time as they have to have various aspects of the software explained.

For some time there have been a considerable number of claims that software *needs* to be produced by multi-disciplinary teams (Axtell, Clegg, and Waterson, 1996; Baeker and Buxton, 1987a; Barnard, 1991; Curtis, Soloway, Brooks, Black, Ehrlich, and Ramsey, 1986; Maxwell, 1996; Norman, 1996). Baeker and Buxton believed that the reason for this is that more skills are required in software production than one person can have, and Norman agreed that the team needs to be comprised of people with extremely different talents. A number of difficulties experienced with multi-disciplinary teams are cited in the literature, examples include:

- at Apple Computer, the merging of industrial design and behavioural design has brought with it the realisation that the people talk a different language (Norman, 1996);

- efforts to involve a graphic designer in a development team took over a year, part of this was due to the need to educate the graphic designer and part was due to the need to break down resistance from developers who had become attached to their designs (Grønbæk *et al.*, 1993);

- now that software is a multi-disciplinary activity, new alliances have formed and misunderstandings naturally arise (Kim, 1990);

- Karat (1996) suggested that the "difficult necessity of multidisciplinary communication in design" should be given some consideration in usability engineering;

- each discipline has its own perspectives and priorities (Erickson, 1990; Kim, 1990) and this can introduce problems of a political nature (Erickson, 1990);

- what one person in a multi-disciplinary team finds valuable, others do not notice, and they do not notice that they have not noticed (Kim, 1990).

Despite the emphasis on multi-disciplinary teams in the literature, there is reason to doubt the widespread existence of such teams in current practice. One reason to doubt the current widespread existence of multi-disciplinary teams is the type of role that HCI and human factors specialists currently play (see section 1.10.1). Practitioners in these disciplines commonly function externally to the team as consultants and advisors. Whilst this mode of working has its own myriad of problems, it could hardly be considered that it represents multi-disciplinary teamworking. When multi-disciplinary teamworking takes hold of software production, it is likely that HCI and human factors would be the first disciplines to be introduced into a software production team. It therefore seems likely that software production teams are generally still dominated by the technical disciplines of programming and IT design.

1.10.4 A New Kind of Specialist Required

A number of researchers have suggested that a new kind of specialist is required in software production. Baeker and Buxton (1987a) believed that this specialist should be trained to understand and improve the ways in which humans interact with computers. Similarly, Norman (reported in, Rheingold, 1990) believed there should be a profession of interface designers, trained in design, cognitive science and programming. Heckel (1991) suggested that as programmers are generally poor communicators, people from outside the field will have to be brought in to software production. Already people with qualifications in the technical and human disciplines are emerging, who can play a full role in design and bring the human factors knowledge into a software team, claim Eason and Harker (1991). A new kind of project manager who would be the team's information architect was foreseen by Eason and Harker (1991). Kim (1990) believed a new profession of "interdisciplinary connectors" would emerge who are skilled at explaining any discipline to any other. Their primary purpose would be to help members of a multi-disciplinary team to communicate with each other.

1.10.5 Summary and Final Commentary

It is evident from the literature that common practices in commercial software development involve HCI designers and human factors personnel functioning in a support role, outside the software team. This often leads practitioners to complain that their involvement in the project is too late. Alternatively, the nature of HCI support groups leads some researchers to believe that the HCI designer's role is complete when the specifications are written, failing to acknowledge the reality of the usability issues which arise only during implementation.

Internal HCI or human factors groups often find the volume of enquiries they receive surpasses their available resources. One way to address this problem would be to assign an HCI specialist to each software team. Some researchers believe that this approach would soon outstrip the supply of such personnel. However, the job market indicates that demand for such people in software development is far lower than for programmers.

Some researchers describe the HCI designers role as like the manager and director of a motion picture, an important aspect of whose role is to maintain conceptual integrity of what is being produced. However, there is evidence to suggest that HCI and human factors people rarely have authority to decide on functionality and are often left out of key decisions. There is also evidence to suggest that often user interface design is not carried out by HCI or human factors specialists.

Many researchers believe that HCI practice and software design involves an element of craft skill, and intuition also seems to be evident in practice.

Although individual differences among programmers have been shown fairly conclusively, the literature contains a large number of generalisations about them. Such generalisations have some use in illustrating the kinds of behaviour and characteristics which can be found in commercial programmers. The views and frustrations of some members of the HCI and human factors community serves to demonstrate the friction which exists between the roles. Despite the widespread acknowledgement of the existence of individual differences among commercial programmers, the consequence of this finding are rarely reported. Because differences exists among commercial programmers, this implies that programmer selection is difficult. This further leads to the conclusion that software teams are usually of mixed ability.

Some of the most significant software ever produced was constructed by teams of two or three people. Some researchers believe that today's software teams are therefore much larger, by comparison with these special cases. This view is felt to be misguided as there have always been examples of both very large and very small software teams. The continued existence of software team structures developed in the early 1970's partly demonstrates this. Perhaps what the research community is trying to say is that we still do not have a good solution to the problem of making a large software team work. Even though sensible team structures have existed for over 20 years, they are claimed to be effective for teams of no more than around 12 people.

Practical experiences of software developments have shown that it is not unusual for at least the early stages of a software development to be dominated by a small number of individuals in the team, and that this not usually a problem. A more serious concern from current practice is that it is not uncommon for design and implementation to be done by separate teams, communicating only via a written specification.

Although the need for multi-disciplinary teams in software production is well established, their widespread existence in commercial software development is doubted. One reason for this doubt is that HCI and human factors specialists currently have a role outside the development team. As these most obvious disciplines for integration within software teams have yet to appear in the majority of teams, it seems unlikely that other disciplines are represented.

A number of researchers have described a new kind of specialist which is required in software teams. The descriptions of these specialists corresponds with the kind of role a HCI or human factors person could perform within a software team.

1.11 Communication and Comprehension within the Software Team

This section begins with an overview of communication, comprehension and collaboration within a software team. The importance of conceptual integrity is described and the existing means of facilitating comprehension and maintaining conceptual integrity are investigated. Finally, tools to facilitate communication, comprehension and shared mutual understanding within the team are explored.

1.11.1 An Overview of Communication, Comprehension and Collaboration

Technical communication within a software production team is vital to project success. Curtis *et al.* (1987) suggested that processes of technical communication and negotiation are crucial but are not adequately acknowledged by existing software process models. They believed that constant technical communication is required in software production and this should be addressed by the process model. Similarly, Erickson (1996) believed that effective communication among members of the software design team is necessary, primarily because design is a distributed social process. Karat (1996) has emphasised the need for HCI research to progress from just focusing attention on user involvement to consideration of, "... the difficult necessity of multi-disciplinary communication in design". Even though multidisciplinary teams are thought to be relatively uncommon (see section 1.10.3), the problem of communication in software has long been known about and almost equally applies to teams comprised of software engineers. One of the main points made by Brooks (1975) is that people and months are not interchangeable, because of communication. Brooks argued that if people are added to a project which is running late, it actually makes the project even later, because of the communication burden this creates on the existing team members.

Issues of comprehension in software teams are less well represented in the literature. Boehm-Davis (1988) suggested that not much research had been carried out into the comprehension and comprehensibility of computer programs. However, in software team term, comprehension of the conceptual constructs underlying the software being developed is more important. All team members must share a closely aligned understanding of exactly what they are creating; in other words, comprehension primarily relates to achieving conceptual integrity (see section 1.11.2). Other aspects of comprehension are also important; for example, the designer's own understanding of the problem.

In studies assessing the comprehension of computer programs, Curtis (1988) discovered that half of the variance in comprehension performance was accounted for by individual differences among the programmers studied. Perhaps a more relevant comprehension test is reported by Mayes, Draper, McGregor, and Oatly (1990), who found that regular users of Mac Write could only remember around 50% of the gross details of the application that were measured. However, when they were asked to recall details of a particular procedure, they were a lot more effective. Norman's (1987) observations on mental models provide some support for these findings. Norman suggested that people's abilities to "run" their models are severely limited and suggested that mental models are unstable, i.e. people forget details of the system they are using. These findings could well be highly relevant to the methods used to support comprehension in a software team. From the findings of Mayer *et al.*, participants were unable to recall even basic functionality when they conceptualised the software in a static way. However, when they mentally ran the software, they could recall a lot more information about it. Therefore, **it seems likely that the best way for software team members to remember all the relevant details of an application they are constructing would be to imagine the proposed software in operation**. Both prototypes (see section 1.11.3.3) and scenarios (see section 1.11.3.1) are believed to be especially effective at facilitating comprehension and communication within a software team; perhaps the findings of Mayer *et al.* (1990) provide some understanding for this.

Further empirical evidence relevant to comprehension was generated by Wasserman (1987), who discovered that the understandability of IT design notations was enhanced by the use of sample screens. This provides some confirmation for Brooks' (1975) belief that the external design of the software under construction is vitally important for facilitating team members' comprehension of the conceptualisation of the software.

Further research into mediums of communication was called for by Curtis *et al.* (1986). Krasner (1986) believed that better co-ordination tools were required to improve communication and information flow in a project. Similarly, Mountfield (1990) said that better ways to facilitate collaboration must be found. Erickson (1995) asked **how can communication be facilitated among the audiences of a design**? The continuing call for answers in this area suggests that solutions have not yet been found. Muller (1993) believed concrete visualisation to be an essential tool for interpersonal communication about design but found this category of communication to be poorly understood.

Maintaining shared understanding and conceptual integrity with teams involving HCI or human factors specialists and programmers is a particular problem. In fact, because multi-disciplinary teams are only just beginning to evolve (as discussed in section 1.10.3), it is likely that such problems will begin to dominate as such teams become more widespread. As early as 1975 Brooks foresaw a team member whose role it was to specify the external appearance of the software and to then share this with the rest of the team. HCI designers and human factors specialists are now poised to take on this role. Communication and comprehension issues between the role Brooks' foresaw and programmers were also described in 1975. Brooks' questioned how one could ensure that every little detail of an interface design is communicated to the programmers, properly understood by them and incorporated into the end product. He also highlighted the fact that countless questions about the user interface will come up during the implementation and that puzzled implementers should be encouraged to ask the user interface designer rather than making guesses. Thus, in 1975 Brooks had a clear view of the kinds of issues that would arise when a role such as an HCI designer was introduced into a software team.

Specific collaboration problems between HCI designers and programmers were also cited by Mantei and Teorey (1988) and more recently by Browne (1994). Perhaps further evidence for the lack of widespread multi-disciplinary teams (discussed in more detail in section 1.10.3) is that this particular collaboration has not received more attention in the literature.

1.11.2 Conceptual Integrity

Conceptual integrity is perhaps best described in Brooks' words:

"*Any product that is sufficiently big or urgent to require the effort of many minds thus encounters a peculiar difficulty: the result must be conceptually coherent to the single mind of the user and at the same time designed by many minds. How does one organise design efforts so as to achieve such conceptual integrity? This is the central question addressed by Mythical Man-Month.*" (Brooks, 1995, p. 256)

Brooks is not the only researcher to have the view that software team members need to maintain a coherent vision and understanding of the product they are developing (e.g., Basili and Reiter, 1981, Curtis *et al.*, 1987, Flor and Hutchinson, 1991, Heckel, 1991, and McCarthy, 1995). However, this factor was ignored (or perhaps a simplification made deliberately) by Norman's (1986) definitions of the various models which are relevant to software design. Norman defined the conceptualisation of the system held by the designer as the 'Design Model', the conceptual model constructed by the user as the 'User's Model' and the image resulting from the physical structure that has been built as the 'System Image'. Thus, Norman believes that the designer's primary task is to construct an appropriate 'System Image'. The designer should want the 'User's Model' to be compatible with the underlying conceptual model, the 'Design Model'. Whilst Norman's model is informative and provides a good illustration of the various conceptualisations which designers and users make and how these fit together, it does not acknowledge that the 'Design Model' needs to be understood by all members of the software team. In commercial software development, in order to construct a coherent 'System Image', all members of the software team must share a coherent vision and understanding of the software under production, or the 'Design Model'.

Although Norman's (1986) views were simplified, they did emphasise the importance of conceptualisation within software production. Brooks (1986) believed the essence of software engineering to be the crafting of conceptual constructs. Because of this he expressed the following view:

"*There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.*" (Brooks, 1986, p. 179).

Brooks (1995) has more recently suggested that crafting the conceptual constructs now constitutes more than half of the total effort of producing a software product.

Curtis *et al.* (1987) suggested that the reality of software development is that one or two team members become the primary conceptualisers for the whole team. They also identified individuals particularly gifted in conceptualisation ability as "Super-Conceptualisers" (further discussed in section 1.10.2.1). Brooks (1986) also emphasised the importance of discovering individuals with such talent and later (1995) reaffirmed that such people should be responsible for the user interface (further explained in section 1.10.2.1). He believed that if software is to have conceptual integrity then someone needs to control the concepts. Other references to the importance of conceptualisation ability have also been made, for example, likening the skills required to design software to those required for making films (this analogy is described in section 1.10.1).

From Brooks' (1995) view, it is apparently suggested that the user interface designer should be the keeper of the project vision and be responsible for ensuring that the rest of the team share this vision, "The architect [user interface designer] forms and owns the public mental model of the product...". Whiteside *et al.* (1988) also believed in the importance of negotiating a shared vision for usability and suggested that usability specifications put common meaning into a project by providing an objective and published vision of what the team is trying to achieve. It has also been suggested by Wilson and Rosenberg (1988) that the interface designer should provide a coherent vision of the interface, containing both the elements of the interface and the glue that holds these elements together.

In order to achieve conceptual integrity during software production, Curtis *et al.* (1987) suggested that a vast amount of communication is required to share understanding. This view is also held by Flor and Hutchinson (1991). However, as Curtis *et al.* (1987) pointed out, the waterfall lifecycle models miss the fact that this vast communication is required.

Not only does the dominant commercial lifecycle neglect the need for significant communication during software production, but the advent of multi-disciplinary teams is beginning to make the communication problem more difficult. Sharing an understanding between similarly skilled software engineers has in the past proved to be a fairly significant problem due to their vast differences in ability. Adding new disciplines which must communicate well and share a vision of what they are all working towards clearly compounds the problem.

Both Brooks (1995) and Curtis *et al.* (1987) believed that maintaining conceptual integrity is of vital importance to software production and resulting software quality. The consequences of failing to share a mutual understanding or conceptualisation of a complex software product can be serious. McCarthy (1995) suggested that such a failure means that "you end up with junk". Software projects are infamous in many respects, going over budget, being late, failing acceptance, and being scrapped. Failure of a team to share an understanding and a tight vision of what they are producing is likely to be a very strong factor in such failures.

1.11.3 Existing Mediums of Communication and Means of Facilitating Comprehension and Maintenance of Conceptual Integrity

This section describes existing mediums of communication and means of facilitating comprehension and maintaining conceptual integrity within a software team.

Representations and notations, and specifications are conventional means of facilitating comprehension. The use of prototypes in this role is a more recent idea thought to have considerable potential, but little research has concentrated on this specific use of prototypes. Using design rationale to facilitate comprehension is another relatively recent idea which has yet to used much in practice.

1.11.3.1 Representations and Notations

Many forms of technical representations used within software design (IT design rather than user interface design) do not provide a particularly useful medium for communication with users. Mantei and Teorey (1988) suggested that representations such as flowcharts and Data-Flow Diagrams (DFDs) do not adequately convey the workings of the system to users. A formal and executable notation called USE transition diagrams was described by Wasserman and Shewmake (1990). They believed this to be an excellent way to describe the intended behaviour of an interactive system to users. They ultimately acknowledged that while the approach seemed successful, the users apparently did not really get a true sense of how the system would work in practice. Their appraisal of the success of the notation is perhaps too harsh, because they reached their conclusions based on the fact that users initially claimed to understand the representation and the design but then asked for changes in what they had approved after the system was built. Often it is not until software is in use that people can really see its potential and fully understand their requirements (see section 1.4.2). It is therefore harsh to blame requests for changes solely on the design notation used. However, it is thought that formal definitions lack comprehensibility (Brooks, 1975).

There is a need to ensure that elements of a design are understandable by users and therefore representations for better communication are an area of potential future research (Wilson *et al.*, 1996). One type of representation which is already widely cited as useful for facilitating communication between designer and user is prototypes (see section 1.7.4.1).

From the perspective of the software team, representations have a number of uses centred around facilitating comprehension, communication and visualisation of various aspects of the software design; this includes aiding the designer to better comprehend and visualise the design themselves. Brooks (1995) believed that there is a need for multiple representations of software structure, each covering a distinct aspect, and even then it should be noted that some aspects do not diagram well at all. Curtis (1988) suggested that different representations highlight different forms of information. It is therefore apparent that no single multi-purpose representation will be found.
Many people express the view that better representations are needed and for various reasons. Several pertinent examples include:

- radical improvements in design quality will not be made until stronger models and representations are produced (Newman, 1991);

- there are no convenient notations for describing the appearance of screen objects and such a weakness in interface representation makes it hard to specify the interface (Newman, 1991);

- Hawkins and Reising (1983) discovered that visual representations can cause interpretation difficulties for the viewer, in other words, a picture paints a thousand words and that's the problem;

- software design can lead to situations where human cognitive limits in working memory, mental calculation, etc. are exceeded and can also overload the representational ability of an unaided decision maker, and this can lead to sub-optimal decisions being made. Complex situations like these would benefit from the use of representation aids which help the decision maker to represent and reason about the decision problem (Zachary, 1988);

- Flor and Hutchinson (1991) believed that an individual's effectiveness at solving a problem depends as much on good external representation as good internal representation;

- Lansdale and Ormerod (1994) believed that one aspect of interface notation is whether the user has to undertake particularly hard mental operations (like translating complex grammatical expressions) in order to comprehend the notation itself.

The abstract and complex nature of software necessitates continued interest in improving representation both to assist the thought processes of the designer and the subsequent communication of their design.

Many researchers have suggested requirements for improvements to representations used in software design. Some examples include:

- a notation which can be understood by designers (both the HCI and IT variety), implementers and managers (Bell and Spencer, 1995). Lim and Long (1994a) similarly shared the view that a common notation has potential benefits which could be exploited;

- a human factors notation should satisfy the requirement of communicability, between designers and users, and among designers (Lim and Long, 1994a);

- a human factors notation should satisfy the requirement of maintainability to facilitate specification and records of design, and is therefore amenable to computer support (Lim and Long, 1994a);

- example driven techniques are better than general guidelines and methods (Bødker *et al.*, 1993);

- Fitter and Green (1979) believed that useful notations should contain not only symbolic information but also perceptual cues. They suggested that a good notation should exhibit the following characteristics: relevance - highlighting useful information to the user; redundant recoding - using perceptual and symbolic characteristics to highlight information; revelation - perceptually mimicking the solution structure.

Some of the more difficult requirements proposed have begun to be addressed. Stories (Erickson, 1996) and Scenarios (Carey *et al.*, 1991; Carroll and Rosson, 1990) have been found to be good design representations by virtue of their communicative, easily comprehensible and common currency representation. This satisfies some of the common currency requirements (Bell and Spencer, 1995; Lim and Long, 1994a) and are by definition, example-driven, fulfilling the requirement suggested by Bødker *et al.* (1993).

Further benefits of scenarios and stories that have been discovered or suggested include:

- scenarios are advocated as a key discount usability technique as a cheap form of prototype enabling the designer to get quick and frequent feedback from the user (Nielsen, 1993);

- in the continuing search for technology to provide formal support, Karat and Bennett (1991) found that simple scenarios have proved most successful for maintaining a user-centred perspective;

- scenarios are good for conveying ideas in design meetings (Carey, *et al.*, 1991) and stories are similarly regarded as powerful tools for discussion and persuasion, even "communication catalysts" (Erickson, 1996);

- a collection of stories is a body of knowledge that can be questioned by all (Erickson, 1996);

- stories are particularly useful for communication because they are memorable and have an informality that is well suited to the uncertainty that characterises much design related knowledge (Erickson, 1996);

- with the recognition that scenarios aid communication, Carroll and Rosson (1990) suggested that their use be stepped up to using them as a representation of design.

It should be noted that there is a distinction between what Erickson describes as 'Stories' and scenarios of use described by others. For the purposes of consideration as a form of representation this distinction is less important as the emphasis is on the simple informal basis which they share. The use of scenarios (and stories) as a representation has clearly gained the support of some key researchers and practitioners. Some purely practical factors are likely to influence the use of scenarios in commercial software production. Firstly, as the inclusion of scenarios in Nielsen's (1993) key discount usability methods demonstrates, scenarios are cheap, practical, simple, powerful and communicative. Secondly, once the approach has been learned, it is completely portable and can be adopted spontaneously on demand. It may be that it is Erickson's mastery of the scenario technique that has led to its extension to the 'Stories' described. Thirdly, **scenarios provide a common representational currency which can be understood by all parties involved in the software production**.

In summary, traditional forms of technical representations of IT designs do not provide a good medium for communication with users. Prototypes are now believed to be a better means of communicating with users. However, multiple forms of representations are thought to be needed to describe the various distinct aspects of software structure. Some conventional representations require hard mental operations just to understand the notation. Problems with using pictorial representations have also been found, as such representations can be misinterpreted.

Some researchers believe that representations need to provide a common representational currency between the parties involved in the software production. Scenarios are believed to be such a technique. They are cheap, effective for conveying ideas, and work well in practice.

1.11.3.2 Software Specifications

From the point of view of the software team, the main purpose of a specification is to communicate software requirements and preliminary designs to all team members in a way which facilitates their unambiguous comprehension.

Miller-Jacobs (1991) suggested that the process of conveying information from writer to specification reader is difficult at best and pointed out that software is built to conform to the reader's interpretation of the specification. Therefore, he believed things get lost in translation between specification and product. Grudin (1993) concurred that the product might not be what the specification writer had in mind. Also of the view that paper specifications are open to interpretation by each individual reader were Wilson and Rosenberg (1988).

Muller (1993) suggested that requirements documents are notoriously unsuccessful at guiding implementers' design decisions. Brooks (1975) believed formal definitions to lack comprehensibility; Grudin (1993) similarly claimed that the written specification is not enough to communicate a product idea in the field of interactive software. It was suggested by Curtis *et al.* (1987) that artifacts produced by specifiers are insufficient to convey all information required by software developers to implement the software. They believed in a need for constant technical communication between the various parties involved with the specification, design and implementation of the software. Curtis (1988) later suggested that bridging the gap between the statement of requirements and preliminary design is a necessary future advance. Other future research areas identified by Curtis *et al.* included ensuring the completeness of the requirements statement and co-ordinating the work of project team members.

It has been suggested that prototypes could be used to support software specification. Gomaa (1983) believed that prototypes can often complement the functional requirements document. Similarly, Wilson and Rosenberg (1988) believed that using prototypes in this way can reduce some of the ambiguity associated with the written word. Heckel (1991) agreed that a prototype can lessen the scope for misinterpretation of such documents. Although Verteleney and Booker (1990) did not specifically cite prototyping, they claimed that a form of visual specification can be especially useful if there is a large group of people involved in the specification process. It has also been suggested that prototypes could be a way to facilitate the iterative specification of requirements (Brooks, 1986).

Olson *et al.* (1990) suggested other means to support software specification. They found that notations to specify the user interface can be awkward and difficult for non-programmers and instead suggested specification by example which removes the problem of encoding into a notational form. Further they suggested specification by "composition" which involves constructing a specification by re-using aspects of old specifications stored in libraries, and also using generic or template specifications which are tailored to suit the current need. Olson *et al.* proposed that a tool to support specification expressing information wherever possible, in terms of its external appearance (hence the emphasis on specification by example). They also proposed that such a tool should also actively support the design of good user interfaces.

In summary, as a document which defines the work of the software team, the primary purpose of a specification is to communicate software requirements and preliminary designs to all team members in a way which facilitates their unambiguous comprehension. However, in practice, conveying requirements from writer to specification reader is difficult and software is usually constructed conforming to the reader's interpretation of the specification. Things commonly get lost in translation between the specification and the product so the resulting software is often not what the specifier had in mind. Written specifications are therefore considered to be insufficient to communicate a product idea in the field of interactive software. There is a need for constant technical communication between the various parties involved in the specification, design and implementation of the software.

Prototypes have been suggested as a means of supporting software specifications by visually and interactively illustrating their content. This is believed to reduce some of the ambiguity and misinterpretation which results from the sole use of written specification documents.

1.11.3.3 Prototypes

Many researchers strongly believe in the potential of prototyping within software production. Cohill (1991) described it as a "critical tool" and Curtis *et al.* (1987), as a "powerful" approach. Prototyping is described by Brooks (1986) as one of the most promising of current technological efforts, as the approach directly addresses the conceptual essence of software production.

This section suggests that "Chauffeured Prototyping" is in widespread use. It then examines evidence for the use of prototypes as a medium for communication, and as a means of facilitating comprehension within the software team.

1.11.3.3.1 Chauffeured Prototyping

Preece *et al.* (1994) coined the expression, “Chauffeured Prototyping”, to describe prototypes where the user watches while another person ‘drives’ the prototype. Preece *et al.* suggested that this is a way to test whether the interface meets the user’s needs without having to examine low level interaction. This form of prototyping technique is described by Anderson and Olson (1987) as “façading” and as a simulation of the external appearance of the software’s interface.

Chauffeured prototyping is not widely reported in the literature, but it is considered to form the basis of techniques which are common in commercial software practice. Evidence supporting this assertion includes the following points:

- “Our evidence suggests that most design teams that are building prototypes are not, at present, using them to evaluate usability. When they are shown to users they are usually used for demonstration purposes.” (Eason and Harker, 1991, p. 86);

- Wagner (1990) suggested that visualisations (including prototypes) are useful for demonstrating a concept even if the user cannot interact with them;

- Wagner also described prototypes which are “hard-wired” and exhibit minimal branching and scripting to make the interface seem realistic;

- “... a scenario depicting the user interface requirements through a sequence of events is devised. The scenario should reflect how the end users would utilise the system since the scenario drives the execution of the user interface prototype and is the mechanism through which the analyst communicates with end users.” (Acosta *et al.*, 1994, p. 68);

- “In the mock-up techniques, the developers help the users to envision the system by providing constructed concretizations of the technology.” (Muller, 1993, p. 227);

- Mantei and Teorey (1988) believed that the reaction of future users to a “mock-up” (a term which does not imply interactivity) will generate information as to whether the software currently envisioned will be acceptable;

- Baeker and Buxton (1987b) also employed the term “mock-ups”.

The term “mock-up” is considered to imply prototypes which are demonstrated rather than interacted with. Similarly, many researchers describe users being ‘shown’ a prototype, such descriptions also imply that prototypes were demonstrated rather than interacted with by the user. Examples of such descriptions include:

- “... the important element [of rapid prototyping] is the visual representation of the system, that is, the user interface... By actually seeing the user interface the functionality of the system becomes much clearer.” (Miller-Jacobs, 1991, p. 277) (emphasis added);

- "Once the users saw the user interface "strawman" implemented [in the prototype] and understood the capabilities of REE's rapid prototyping facilities, they immediately had additional suggestions for improvements to meet their complex needs." (Acosta *et al.*, 1994, p. 72) (emphasis added);

- "...showing users prototypes does not, of itself, guarantee a usability." (Trennor, 1995, p. 5/1) (emphasis added).

- Miller-Jacobs (1991) also acknowledged a difference between an interactive prototype and a simulation, apparent from a description of a prototype which was constructed, consisting of interactive and simulation aspects.

Although the evidence of the widespread use of chauffeured prototyping is not explicit, the way that prototypes have been described by many researchers does suggest that such prototypes are common.

1.11.3.3.2 Prototypes as a Medium of Software Team Communication

There is some suggestion in the literature that prototypes have a use as a medium for communication within the software team. However, this appears to have only been superficially explored by researchers. Evidence to support these claims includes:

- prototypes can help to facilitate communication in the software team (Preece *et al.*, 1994);

- a fully functional user interface prototype can serve as a dynamic communication tool (Wilson and Rosenberg, 1988);

- Miller-Jacobs (1991) believed that the visualisation of system requirements afforded by prototyping conveys far more information (than written documentation) to the system developer;

- Wilson and Rosenberg (1988) suggested that prototypes provide a common reference point for all members of the design team (as well as users and marketing);

- prototypes and simulations provide a medium for conveying the user specification to IT designers (Damodaran, 1991);

- " [the prototype enables the team] ... to get straight to the heart of concerns, as everyone has a clear concept of what the team is attempting to do. With easy access to the history of visualisations and interactive prototypes, controversial areas are resolved more quickly and to everyone's satisfaction." (Wagner, 1990, p. 82);

- the use of prototypes can reduce dependence on verbal communication (Glushko, 1992).

Some researchers have taken the claims that prototypes can facilitate communication within the software team, one step further. They have begun to suggest that prototypes can be used as a way of capturing project knowledge and answering questions about the proposed software. Examples of such comments include:

- a prototype provides a common database for those involved in the project (Heckel, 1991);
- a prototype provides a means of testing product specific questions (Wilson and Rosenberg, 1988);
- an important benefit of prototyping is the speed with which new team members can be integrated as a result of the history portrayed by the visualisations (Wagner, 1990).

Of particular interest is the view of Carey *et al.* (1991), who suggested enquiries that software engineers make of human factors specialists about a specific system are best supported by context sensitive access from the exemplar system or from a mock-up. The use of prototypes to support communication between human factors specialists (including HCI designers) and software engineers has otherwise not been explored in the literature.

If prototypes are to be used as a communication medium, Wagner (1990) suggests that they should be easily distributable (i.e. accessible) and contain comments windows to allow for feedback.

The current lack of appreciation for the use of prototypes for facilitating communication and comprehension within the software team is perhaps best illustrated by Kinmond’s (1995) survey into the use of prototyping which apparently contained no consideration of these benefits. Specifically, participants were asked to rank a given list of benefits of prototyping which did not feature items relating to communication or comprehension.

1.11.3.3.3 Prototypes as a Means of Facilitating Comprehension

As well as facilitating communication within a software team, prototyping is also considered to facilitate various aspects of comprehension. Aspects of comprehension which prototyping is thought to facilitate include:

- helping the HCI designer to better visualise, comprehend and reflect on their own user interface design;
- facilitating the sharing of understanding of the proposed software by members of the software team, i.e. maintaining conceptual integrity (Myhill and Brooks, 1996);
- facilitating the retention of understanding of the proposed software as the team composition fluctuates over the life of the software production.

As with the literature relating to the use of prototypes as a medium of communication, there has been no particular focus on the use of prototypes to facilitate comprehension. However, some researchers clearly believe that prototypes also have potential in this area, and examples include the following:

- "[prototyping] provides a common frame of references for developers... One of the most difficult parts of any development program is to ensure that accurate information is conveyed between the system specifiers and the developers. Even among developers there may be misunderstanding, as each has a different discipline and frame of reference." (Miller-Jacobs, 1991, p. 279) (emphasis added);

- prototyping enables the designer to make explicit the differences in understanding that exist within a design team and it enables the designer to resolve these in a communicative context (Wilson and Rosenberg, 1988) (emphasis added);

- a project proposal is more open to interpretation than a prototype (Heckel, 1991) (emphasis);

- Gladden (1982) believed that a physical object (like a prototype or mock-up) conveys more information than a written specification, and suggested that "a picture paints a thousand words";

- Rudd and Isensee (1994) concurred, suggesting that, "A picture may paint a thousand words but a prototype is worth a thousand pictures", in the context of using the prototype as a specification for developers to code to.

Thus, the prototype can become a tool to assist the articulation of the designers' mental conceptualisation to the rest of the team as a means of maintaining conceptual integrity. A stronger indication of the recognition that prototypes support people's understanding of proposed software comes from the body of research suggesting that prototypes have a role in specifying software.

Some researchers have claimed that a prototype can be used in place of a specification (e.g. Rudd and Isensee, 1994; Wagner, 1990; Wilson and Rosenberg, 1988).

Wagner (1990) and Wilson and Rosenberg (1988), however, have also suggested using prototypes to complement a written specification. This more moderate view has received support from Brooks (1986) and Gomaa (1983). A similar view has been expressed by Ehn (1993), who believes that many aspects of a prototype or mock-up cannot be explicitly described in a formal language. Wilson and Rosenberg also claimed that using a prototype in this way can reduce some of the ambiguity associated with the written word.

Wilson and Rosenberg (1988) also believed that prototypes could improve the quality and completeness of a specification and suggested that when novel software is being developed, the prototype may even help to generate the specification. Overmyer (1991) also believed revolutionary prototyping to lead to better a specification than occurs by usual means (i.e. via the waterfall lifecycle). Similarly, Gomaa (1983) suggested that the use of a prototype may lengthen the requirements document as it would be more complete.

The acknowledgement of researchers who believe that prototyping can complement specification also implies that they believe that prototypes can be used to facilitate comprehension within the team.

One further concern with using prototypes to facilitate comprehension is the naivety with which the approach has on occasion been advocated. For example, with respect to prototypes and mock-ups, Gladden (1982) suggested that a picture paints a thousand words. Rudd and Isensee (1994) added that a prototype is worth a thousand pictures. What these views fail to acknowledge is that if the prototype is conveying so much meaning, there is implicit scope for misinterpretation. This is illustrated by the study reported by Hawkins and Reising (1983) (see section 1.11.3.1), which concluded that a picture paints a thousand words, and that is the problem. As Rudd and Isensee suggested, prototypes are richer in meaning than single pictures. In other words, a picture may paint a thousand words, which can be a problem but a prototype paints a thousand pictures, and that can be a serious problem. The negative aspect of misinterpretation when prototypes are used to facilitate comprehension has not been reported in the literature.

1.11.3.3.4 Summary and Final Commentary

Although chauffeured prototyping is not widely reported in the literature, it is believed to form the basis of techniques which are common in commercial software practice. Primary evidence for this belief are descriptions of the prototypes in the literature, including prototypes developed to demonstrate concepts, the term mock-up and the description of users being "shown" prototypes. Such descriptions imply that the user is not interacting with the prototype and it is being demonstrated by a "chauffeur". Although there are some suggestions in the literature that prototypes have a use as a medium for communication within the software team, this has only been superficially explored by researchers. There is even less literature on the subject of using prototypes to facilitate comprehension within a software team and using it to maintain conceptual integrity, although some researchers appear to recognise this potential. Furthermore, some researchers appear to take a naive view of prototypes suggesting that "a picture paints a thousand words". Such researchers fail to realise that a thousand words not written down leaves considerable scope for misinterpretation.

1.11.3.4 Design Rationale

Design rationale is already routinely captured in the memories of developers, minutes of meetings, memos, specifications, etc. (Conklin and Burgess-Yakemovic, 1996). Grudin (1996) concurred and suggested that the weakness of this approach is the incomplete, unstructured and distributed nature of such stores of design rationale. Further, he suggested that such records are hard to retrieve and interpret at a later date and can be inaccessible to third parties. Furthermore, artifacts produced by some design process (for example, prototypes) do not indicate the reasoning (i.e. the motivations, requirements, constraints, negotiations, etc.) underlying its design (Moran and Carroll, 1996). Thus, although thought to be useful, design rationale is often lost and only the resultant artifacts remain.

Researchers have attempted to provide more formal means for the capture of design rationale. However, they have been little used in practice (McKerlie and MacLean, 1993). Grudin (1996) has recently stated that convincing examples of the successful use of design rationale in practice are required to prove that its capture is worthwhile and to allay fears that capturing it will create too much work and slow projects down. Concern has also been expressed by Buckingham-Shum (1996) that representing design rationale may prove to be too much work during the design process as creating formalisms for design rationale inevitably creates work for designers.

A number of techniques for recording design rationale have been proposed. Probably the most well known of these is QOC (see MacLean, Young and Bellotti, 1996), a semi-formal representation, which stands for 'Questions, Options and Criteria'. Design rationale is expressed by first considering the design question, then the available design options and finally the criteria which the design should meet. Quite complex design rationales can be constructed using this fundamentally simple idea by interlinking design questions, their options and criteria. In practice, it is easy to apply, because the underlying philosophy of the technique is conveyed in its name. Furthermore, QOC facilitates the consideration of design options and criteria from all disciplines represented in the design team, on an equal basis. Another reason to think that QOC has potential are the beliefs expressed by Carey *et al.* (1991) that design rationales are best expressed as questions. Carey *et al.* used an adapted form of QOC in a software tool they developed called DRaHFT, which stands for 'Design Rationales and Human Factors Transfer'. The objective of this tool was to provide software engineers with an understanding of human factors through the use of design rationales and scenarios. Carey *et al.* (1991) believed that enquiries made by software engineers are best supported by direct access to an exemplar system or a mock-up. They described a prototype of DRaHFT where screen mock-ups are presented with hypertext links to the design rationale information which they illustrate, and suggest that in a real DRaHFT tool, the exemplar system could be linked directly to the information in DRaHFT. The basic idea also encompassed the concept of keeping a library of user interface exemplars (which could also store information about subsequent evaluations of the designs), which could help software engineers in the selection of appropriate widgets for a user interface (Carey, Ellis, and Rusli, 1993).

Other means for recording design rationale have also been explored. Scenarios are thought to have some utility in this regard (Carey *et al.*, 1991). Muller (1993) described a project where a video diary of design meetings was kept as a means of communicating design rationale with implementers.

One of the main reasons for capturing design rationale is that it is considered to be a good way for communicating within the software team (Carey *et al.*, 1991; Harker, 1991; McKerlie and MacLean, 1993). Grudin (1996) suggested that design rationale could generate major productivity gains by assisting in the education of new arrivals to a software team and freeing up existing team members to continue their work. He claimed that design rationale could become a means of retaining project memory.

In addition to its potential use in communication within the software team, design rationale is thought to be a way to facilitate comprehension of design decisions made. Carey *et al.* (1991) believed that design rationale provides a means for making human factors decisions understandable to software engineers which could help them to understand the contribution of HCI. The QOC design rationale representation is partly designed to clarify the contribution of parties from different disciplines to the design decisions (Bellotti, 1993). Lewis, Rieman and Bell (1996) believed that abstracting to general design rationale representations would be too distracting for software developers. This view was apparently shared by Carey *et al.* (1991), who suggested that in order to be understood, design rationale needs to be considered in context, that is, the context of the designed artifact it applies to. Carey *et al.* advocated linking design rationale to an exemplar system, because they believed that software engineers need to interact with the system, at the same time as considering design rationale, in order to understand it.

Another advantage of formalising the recording of design rationale is that this leads to more intensive scrutinisation of the design by the designers themselves (Buckingham-Shum 1996; Carey *et al.*, 1991; MacLean, Young and Bellotti, 1996), enabling them to critique and improve their designs.

Overall the research community appears to be uncertain of the practical utility of capturing design rationale. Some are concerned about the time it would take designers to formalise design rationale (Buckingham-Shum, 1996; Grudin, 1996). Grudin expressed further concern that design rationale could become a record of failure and believed that such failures are more productively forgotten than over-analysed. Carey *et al.* (1991) state that the premise that software engineers can understand, and work with design rationale, also needs to be tested.

Therefore, although design rationale is already routinely captured during software projects, it usually exists in the heads of designers and in project documentation. Formally recording design rationale is rarely undertaken, although the research community have come up with some potentially useful ways of representing it. Records of design rationale are considered to be a means of facilitating communication and comprehension within a multi-disciplinary software team. Such design rationales are believed to make more sense (by being less abstract) when they stand alongside an exemplar or mock-up of the system they relate to. Further benefits result from the close scrutiny of the design, required to record design rationale, which can lead the designer to make improvements to the design. The research community is apparently undecided as to whether the efforts required to record design rationale will be worthwhile.

1.11.3.5 Summary and Final Commentary

Traditional forms of representation and notation do not provide good mediums of communication. However, the use of scenarios as representations have been found to be a cheap, practical and effective communication medium.

From the perspective of the software team, the primary purpose of specifications is to communicate software requirements and preliminary designs to all team members in a way which facilitates their unambiguous comprehension. They have been found to be insufficient in fulfilling this role. Because of misinterpretation and ambiguity, software is often produced which was not what the specifier had in mind. There is a need for constant technical communication between parties involved in the specification, design and implementation of the software. Prototypes have been suggested as a means of supporting this communication.

Chauffeured prototyping is believed to be widespread in commercial software practices. Some researchers have begun to suggest that prototypes can provide a medium of communication within the software team, but this has not been explored in any depth. Similarly, using prototypes to facilitate comprehension and maintenance of conceptual integrity of the product within the software team is hinted at in the literature but is not explored. Some researchers have a naive view of prototypes used to facilitate software team collaboration, suggesting that they must be useful because, after all, "a picture paints a thousand words".

Design rationale is already routinely captured during software projects usually in the heads of designers and in project documentation. However, formally recording design rationale is rare. Researchers believe that such records would be a useful means of facilitating communication and comprehension within a multidisciplinary software team. Design rationale is thought to have greater potential when it is used alongside an exemplar or a mock-up as this makes the design reasoning represented less abstract. Researchers are still undecided whether the utility of design rationale in commercial software development is worth the price of formally recording it.

1.11.4 Tools to Facilitate Communication, Comprehension and Shared Mutual Understanding in Software Teams

Long before the widespread adoption of computers, Engelbart (1963) surmised that the accumulated knowledge of humanity has exceeded our ability to handle it. He suggested that only by "augmenting man's intellect" (with the use of computers) could we address the situation. His paper strongly emphasises the use of computers to aid comprehension, he said,

"*By "augmenting man's intellect" we mean increasing the capability of a man to approach a complex problem situation, gain comprehension to suit his particular needs, and derive solutions to problems. Increased capability in this respect is taken to mean a mixture of the following: the comprehension can be gained more quickly; that better comprehension can be gained; that a useful degree of comprehension can be gained* *where previously the situation was too complex.... We do not speak of isolated clever tricks that help particular situations. We refer to a way of life in an integrated domain where hunches, cut-and-try, intangibles, and the human "feel for a situation" usefully coexist with powerful concepts, streamlined terminology and notation, sophisticated methods, and highly-powered electronic aids..."* (Engelbart, 1963; p. 1)

This section will investigate the tools that have been applied to the complex domain of software production to facilitate communication, comprehension and shared mutual understanding within software teams.

1.11.4.1 Tools to Facilitate Visualisation, Conceptualisation and Comprehension

Aside from prototyping, tools to help software designers and developers visualise or conceptualise are scarce. However, the need for tools to help designers visualise is recognised by some. Mountfield (1990) advocated that user interface designers should think about how to create tools to better build on their knowledge and make it reusable to others. In her view, the best contribution that user interface designers could make to understanding the communication interface, was to create and build better tools to do their own jobs.

Supporting designers' "conceptualisation power", particularly their ability to visualise, was felt by Woods and Roth (1990) to be the best help that designers could be given when facing ill-defined problems. Woods and Roth believed that the most important contribution of Artificial Intelligence to decision support may in the long run turn out to be in the form of "...cognitive tools that amplify human powers of conceptualisation" (Woods and Roth, 1990; p.25).

1.11.4.2 Tools to Facilitate Communication and Shared Understanding

Engelbart (1982) recognised the need for a coherent community of augmented individuals to form into augmented teams. Engelbart believed that collaborative communication capabilities were required to realise this, in his words,

"...the synergistic effect of integrating many augmented individuals into one coherent community makes each element more valuable than if it were applied just to support one individual - this is derived from the collaborative communication capabilities....to integrate the augmented capabilities of individuals into augmented teams and communities."

Other than the use of prototyping in this mode, tools to facilitate communication and shared understanding in the software product domain are scarce. Waern (1988) suggested that one purpose of using Computer-Aided Design (CAD) software in engineering is to communicate the results of the design work. CASE tools do not facilitate the communication or shared understanding of the products of software engineering in a form, which is as palatable and easy to understand to a diverse range of individuals, as the end results of a CAD project. Visualisations afforded by prototypes have much more in common with the end results of a CAD project than other forms of representation in the software production domain.

Although tools to facilitate communication and shared understanding are rare, their need does have some limited recognition. Krasner (1986) suggested that augmenting the information communication network with better co-ordination tools would improve flow in a software project. Perhaps such tools are beginning to emerge in the form of electronic mail and intranet, but it is difficult to conceive how even this technology can begin to convey an understanding of the details of a complex software venture sufficiently well to enable diverse individuals in software teams to share an understanding of what they are producing. Such new technology is still predominantly text-based and is therefore subject to similar problems associated with written specifications and other text-based representations used in the software process. One key advantage with intranet and electronic communication is that it is immediate, so team members can be kept up-to-date with events and changes to the software under production much more effectively than with voluminous paper-based documents.

1.11.4.3 The Potential of Hypermedia Tools

Researchers believe that multimedia offers an excellent and efficient means of improving the quality, delivery and presentation of educational and informational material (Scuprowicz, 1990). Hypermedia is similarly claimed to integrate the best characteristics of a wide variety of approaches to learning, thus the presentation of information in this way suits a wide variety of learning styles (Perzylo, 1993). Elkerton (1988) claimed that if online help is designed with cross-references and indexes, it can enhance users' capability to retrieve information quickly. He also claimed that diagrams and animation may help users understand the user interface structure. Harland (1989) suggested that hypertext documents can support effective exploration of a problem space described therein.

One key advantage of hypermedia in the domain of software production is that team members using the media are not disadvantaged by their lack of knowledge when consulting colleagues. This benefit of hypermedia is recognised by Perzylo (1993). Eberts and Brock (1988) claimed similar benefits exist in Computer-Aided Instruction (CAI) software, where students are able to succeed or fail in private.

Clear disadvantages also exist when hypermedia is used for communication and learning. One is that modes of communication between the user and the computer are limited (Hartley, 1980). This leads to another difficulty, which is that CAI (and hypermedia) requires careful and skillful authoring to anticipate the responses and requirements of the student/reader (Eberts and Brock, 1988).

1.11.4.4 Summary and Final Commentary on Tools to Facilitate Communication, Comprehension and Shared Mutual Understanding

Although the need to facilitate comprehension was foreseen as long ago as 1963, few tools exist to help members of software teams to visualise and comprehend the software they are producing. Tools to facilitate the sharing of understanding (or vision) within a software team of the software being produced are not available. A small number or researchers and practitioners believe that prototyping has a role to play both in helping the individual to visualise and helping a team to share an understanding of the software they are producing. There is no research into this specific usage of prototypes in software production. Other than prototypes, it would appear that hypermedia tools may have advantages for communication and comprehension within software teams. The most important of which is the recognition that hypermedia users need not feel threatened by the perceived triviality of information they require from the software, as they might feel threatened when asking similar questions of colleagues. A benefit such as this recognises and works with the people-oriented nature of software production.

1.11.5 Summary and Final Commentary on Communication and Comprehension within the Software Team

The importance of communication within software teams is well recognised in the literature. There has been less emphasis on issues surrounding aspects of comprehension within the software team. However, a number of eminent researchers have emphasised the fundamental importance of maintaining conceptual integrity of the design within the software team, which is considered to be one key aspect of team comprehension. Conceptual integrity has not received the level of attention in the literature consistent with the fundamental importance which eminent researchers believe it to have.

The utility of contemporary forms of representation and notation and specification and prototyping as mediums of communication, and as means of facilitating comprehension and maintenance of conceptual integrity within the team have been explored. Scenarios are believed to have some potential as a medium of communication. Written software specifications are not considered to be a useful communication medium, although they are commonly used as such. Some researchers have suggested that chauffeured prototypes are an effective medium of communication within the software team. However, this has not been explored in the literature in any depth. Producing records of design rationale has been considered as a means to improve collaboration within software teams, but researchers are apparently undecided as to whether such records would produce a net benefit.

Tools to facilitate communication, comprehension and shared mutual understanding within a software team are not available. Prototyping appears to have the most promise in this regard.

1.12 Summary of Findings from the Literature

This summary highlights the current conditions in software production and positive future directions, some of which have yet to be thoroughly researched.

1.12.1 Current Software Production Conditions

A number of aspects of the literature review describe the current conditions of software production. The first point to notice from the review is that there is clearly something wrong with the way that software is produced. A large proportion of software development projects fail through problems in production and during acceptance.

The continuing dominance of the waterfall lifecycle is believed to be a fundamental part of the problem in software production. The process has long been regarded as a poor facilitator of software production. Doubt has been cast on many of the assumptions that the waterfall lifecycle depends upon. For example, a key milestone, which is widely believed to be unrealistic, is the production of a specification document early in the production process (before any implementation has taken place). This particular milestone often forms part of the contract between software supplier and client which builds an inherent problem into the remainder of the software project. It is argued in this review that the waterfall lifecycle survives because it provides answers, although they are often ill-founded. The mistaken assumption that it is possible to write a comprehensive specification at the outset of a software project, at least provides something on which to base a contract or tender. An early specification allows resource, time and cost estimates for a project to be produced. Although the basis for this estimation is highly questionable, the waterfall lifecycle provides some means to generate such figures. Other lifecycles, recognised as more appropriate to software production, are less geared to producing such figures, because they acknowledge that to do so would be unrealistic and misleading. Alternative lifecycles often require a different way of working and a different way of procuring software.

The legacy of the waterfall process is a specification document written for a multitude of discrete purposes, e.g. both as a contract and a description of the software for the team to work to, which does not acknowledge the nature of software production. Such written specifications are considered to be a poor medium of communication within a software team, as are many other contemporary software engineering representations and notations.

Some software production problems come about by the nature of the various activities and people involved in the process. Software design and development is considered chaotic and highly susceptible to changes; software teams also contribute to the nature of software production. Significant individual differences in the performance of commercial programmers is well documented, but the implication that programmer selection is difficult and therefore mixed ability teams are the norm, is not reported. Informal roles occurring in software teams also contribute to the nature of software production. Practical experiences report that it is not uncommon for the early stages of a software project to be dominated by a few individuals, and that this is not a problem. However, the inherent nature of commercial software production does conspire to prevent software developers learning from the products of their labours. Developers often come to a project as the programming phase begins and leave when it ends; they may never be made aware of inadequacies in their software, because they have moved on the next project when the support calls start to arrive. Therefore, it is believed that the inherent nature of software development shapes the reality of software production. Logistics are also beginning to worsen this situation, as it is now not uncommon for design and implementation to be carried out by separate teams (sometimes in different countries).

The foothold that HCI has attained in commercial software production is still poor. User interface design is often *ad hoc* and carried out by programmers. The level of user involvement in the design of user interfaces is alarmingly low, in fact, commonly non-existent. It is apparent that industry still does not understand the value of HCI and human factors. This fact can be quite clearly seen by the lack of HCI jobs in software projects compared with programming jobs. Researchers concur with this view, commonly expressing the belief that there is a theory of HCI (against the view that HCI design is a craft-based discipline) but that it is rarely applied in practice. Furthermore, although the need for multi-disciplinary software production teams is well recognised, **the usual consultancy style involvement of HCI designers and human factors engineers (on the side lines)** in software teams provides a clear indication that integrated multi-disciplinary teams are not yet a reality.

One reality of the current software production conditions is the advent of Graphical User Interfaces (GUIs) and the fact that these user interfaces require a larger proportion of the total development effort to produce – even without HCI design consideration. This inherent increase in effort necessary to produce GUIs has not led to a similar improvement in the quality of user interfaces; many are still widely considered to be poor.

1.12.2 Positive Directions

A number of positive directions in software production have come to light through the literature review. Some of these have received little research to date.

Iterative software design and development is believed to be a better method of facilitating software production and is widely thought to have great potential, particularly because it acknowledges the nature of software. However, its use is clearly not widespread. In this literature review, it is argued that this is due to a lack of commercial credibility, hence the continued dominance of the inappropriate waterfall lifecycle. Some iterative development processes have recently been revived, notably Rapid Application Development (RAD). Even its greatest proponents recognise it as having limited applicability, for example, requiring very special client-supplier relationships and flexible contract arrangements.

The major reason for the re-emergence of RAD is the latest breed of software tools that readily facilitate an iterative prototypical approach, for example, Microsoft's Visual Basic. However, the prototyping activity itself has generated some research and commercial interest, divorced from its role as a vehicle for evolutionary software development. Prototyping is reported to bring about changes to software design earlier in the production process (even when not shown to users) – possibly because the approach helps software development team members to better visualise the proposed software. It is also recognised as a good way to capture user requirements, because it provides a common representational currency between the developer and the user. For these and other reasons, prototyping is believed to be a means by which HCI specialists could take on a more dominant role in software production. One particular form of prototyping that is believed to be in common use in software production is 'chauffeured prototyping'. **The use of such prototypes is often implied in literature but rarely stated or explored in any depth. It has been suggested that such prototypes could provide a medium of communication within the software team, but this aspect of their use has not previously been explored**. Few downsides of using prototypes are reported in the literature. Those that have been include the creation of unrealistic expectations within prototype audiences and problems associated with managing the prototype development process. This lack of reported negative effects is likely to indicate a general lack of research in prototyping rather than a genuine lack of negative effects.

The importance of communication within the software team has been well recognised in the literature, but there has been less emphasis surrounding aspects of comprehension within the team. A number of eminent researchers have emphasized the fundamental importance of maintaining conceptual integrity of the design within the software team. This sharing of a vision of the software being produced is possibly the central aspect of comprehension within the team. However, some researchers have expressed surprise that conceptual integrity has not received the level of attention in the literature consistent with the fundamental importance it is believed to have. One means of improving collaboration within software teams that has been proposed, is the production of records of design rationale. Currently researchers are undecided as to whether such records would produce a net benefit. **Tools to facilitate communication, comprehension (including conceptual integrity) and shared mutual understanding within the team are not reported**, although prototyping appears to have promise in this regard. However, lessons from other software tools that aimed to improve team collaboration are well documented. Integrated Project Support Environments (IPSEs) and Computer Aided Software Engineering (CASE) tools have had limited success. It is believed that this is because they have attempted to impose structure on the software production process, which is an inherently chaotic and people-orientated process.

Positive direction from HCI theory comes from an increasing recognition of the difficulty of applying HCI research to software production practice. Some researchers believe that HCI theory has no applicability to the practice of software design, others suggest that HCI design is craft-based. Such views are causing some human factors and HCI techniques to emerge which attempt to improve commercial software production practices, whilst acknowledging the context in which they must be accepted.

1.12.3 The Focus of This Research

This literature review has described two aspects of software production. The first covers the current software production conditions that exist and seeks to describe the realities of software production. Key 'realities' are argued to be:

- that there is something wrong with the way software is produced causing many software projects to fail during production and acceptance;
- that the inappropriate waterfall lifecycle is firmly established throughout the industry for commercial reasons;
- that software production is strongly influenced by the very nature of software development and the people involved in it;
- that HCI has yet to attain a significant foothold in mainstream commercial software production;
- that HCI designers and human factors engineers, when they are employed, are rarely software team members, instead they perform a consultancy role from the sidelines.

The second aspect of this review highlights positive directions in software production. Key positive directions are:

- identification of the potential of prototypes for improving software in various ways;
- identification of the need for comprehension within the software team in order that a shared mutual understanding (and vision) is achieved and conceptual integrity preserved;
- recognition of the fact that much HCI theory is difficult to apply in practice and the suggestion that HCI design could be regarded as a craft-skill.

This research aims to explore the positive directions identified and to take a pragmatic look at the software production conditions argued. In particular, the research will explore the effects of introducing an HCI designer into a software team as a full team member, rather than as a consultant on the sidelines. The research will include consideration of aspects of comprehension within the software team, following the introduction of the HCI designer role. The use of prototypes is perhaps the most concrete positive direction identified by this literature review. This research will therefore focus on the use of prototypes within software teams - an aspect of their use not widely reported.

Chapter 2 **Qualitative Investigation**

Chapter 2 **Qualitative Investigation**

2.1 Introduction

This chapter describes a longitudinal qualitative investigation of the effects of introducing the role of HCI designer into a commercial software team. As a participant-observer, the researcher performed the role of HCI designer in a software team over a two-year period.

2.2 Research Question

The literature review identified current software production conditions and outlined existing key ‘realities’. It then described emerging positive directions which appear to make a genuine contribution to software production whilst acknowledging these ‘realities’:

- Prototypes have diverse potential for improving software production.
- The concept of ‘comprehension’ and maintaining conceptual integrity within the software team is important.
- Much HCI theory is difficult to apply in practice and HCI design can be considered a craft skill.

The amount of research received in these areas to date is small or not appropriately directed towards solving HCI design problems with the software production domain:

- Certain aspects of prototyping have received research interest, for example, it is now accepted that prototypes are useful for capturing user requirements. However, papers describing such prototypes often fail to make it clear that they are describing a “chauffeured prototype” (Preece et al, 1994). Such papers allude to such prototypes with phrases like, ‘the user was shown the prototype’, in other words, the user watched someone else demonstrate it. Such prototypes have received very little research interest and their utilisation as a communication medium within a software team is not evident from the literature.

- Conceptual integrity within the team, or maintaining a shared comprehension of the artifact being produced, has received little research. Fred Brooks described the importance of this aspect of team collaboration in 1975 and more recently expressed surprise at the lack of research interest (Brooks, 1995). This is a major research gap because the implications of research into the individual differences in computer programmer performance imply that commercial software practice involves mixed ability teams.

- The difficulty of applying HCI theory in practice has received research interest. However, much of this fails to address the ‘reality’ that HCI professionals usually operate outside software teams as consultants. To assess the postulation that HCI design may be a craft skill, it is necessary to analyse such crafts-people in an appropriate setting, where they are in control of their craft, rather than advising others on it (in other words, where HCI designers are full team members rather than consultants).

In order to explore these positive directions and fill the gaps in existing research a research question was formulated, based on one of the key ‘realities’ argued, that would allow exploration of each of the positive directions.

Research Question

What are the effects of introducing an HCI designer into commercial software projects as a full team member?

Prerequisite

The HCI designer must bring with them knowledge of prototyping techniques and HCI theory, and may use these as appropriate in designing the HCI of the software.

The research question and its prerequisite will allow the application of HCI in practice though the creation of a new role in a software team to be explored. The effect of creating this role cannot be predicted, because no research available describes such a role. The prerequisite provides a guide to the focus of the exploration without forcing artificial constraints on the HCI designer. As the research question states, the software project selected for this study must be a genuine commercial venture, encountering real-world constraints and tensions.

2.3 Characteristics of the Research Design

From the findings of the literature review and the stated research question a number of required characteristics of the research design are evident.

Firstly, there is a genuine lack of research in the three key areas being explored by the research question:

1. The diverse potential of prototypes within a software team has not been fully explored.
2. The concept of comprehension/conceptual integrity within software teams has received little research interest.
3. There are no reported studies of HCI designers working within software teams.

This lack of established findings in these key research areas precluded the formulation of an *a priori* research design. The research design had to cater for the exploratory nature of the research question.

The second required characteristic is the need to ground the research in the ‘real world’ of commercial software development, because HCI theories developed in the abstract have been shown to be of little practical use.

Thirdly, the research design had to cater for the fact that there are no reported studies of HCI designers functioning as full team members. To carry out the research, a suitable team needed to be found.

2.4 An Exploratory Research Design

The characteristics of the research design immediately discount experimental and survey research strategies (using Robson's (1998) definition of the division of research strategies into experimental, survey and case study). The exploratory nature of the research called for the rejection of an *a priori* approach that would attempt to isolate well-understood constructs and measure their interrelationships. Yet a further reason for the rejection of the survey strategy comes from the revelatory (Yin, 1994) nature of the research. In other words, finding a sample of software development 'cases' where an HCI designer was acting as a full team member was not considered a viable proposition as this situation is considered extremely rare.

The lack of reported instances of HCI designers functioning as full team members discounted the possibility of any form of large scale study. It followed that small scale, in depth research designs would be more appropriate.

Miles and Huberman (1994) suggested that exploratory in-depth research usually contains some or all of the following criteria:

- Intense and prolonged contact in the field.
- Designed to achieve a holistic or systemic picture.
- Perception is gained from the inside based on actors' understanding.
- Little standardised instrumentation is used.
- Most analysis is done with words.
- There are multiple interpretations available in the data.
- The challenge is to find the most compelling interpretation based on theory or internal consistency.

These criteria are satisfied by a longitudinal qualitative investigation.

2.5 Investigation of Potential Qualitative Research Strategies

Having been directed towards a qualitative approach, this section investigates potentially appropriate qualitative research strategies.

2.5.1 The Case Study Research Strategy

This section begins by defining the term case study as a research strategy. The strengths and weaknesses are related to those of an ethnographic strategy. Finally, case studies in the domain of software production are briefly discussed.

2.5.1.1 Overview of the Case Study Research Strategy

The term case study has been used to mean various different things (Robson, 1998). Yin (1994) suggests that a common flaw in social science texts has been to confuse case studies with ethnographies, with participant-observation or with qualitative research (p12). The style of research relying on an observational approach involving a relationship between the researcher and the researched has been described as fieldwork, ethnography, case study, qualitative research, interpretative research and field research, according to Burgess (1984). He finds that these terms are often given different emphasis and meaning according to the discipline of the researcher, amongst other things. Hammersley (1996) concurs that the term case study overlaps with others and adds participant observation, life history method, ethnogenics and ‘etc.’ to Burgess’ list. He also states that these other terms are not used in precisely defined ways either. An entirely different and very broad definition of case studies is offered by Stake (1994) who considers them to be concerned with a choice of object to be studied, rather than a methodological choice.

Both Robson (1998) and Yin (1994) present a solid argument for case studies to be regarded as a research strategy. Robson describes it as:

“Case study is a strategy for doing research which involves an empirical investigation of a particular contemporary phenomenon within its real life context using multiple sources of evidence.” (p5)

Yin (1994) goes even further, suggesting that:

“The Case study is an all-encompassing method – with the logic of design incorporating specific approaches to data collection and to data analysis. …[the case study is] a comprehensive research strategy.” (p13)

Robson’s definition of the case study as a strategy is the one that has been adopted for this research. This strategy does not impose methods of its own design as Yin attempts to. Rather, Robson’s deliberately leaves open the particular methods that will be selected for carrying the research within the framework of the case study strategy.

Robson (1998) asserts that the flexibility of design demanded by an exploratory research question provides a strong case for the adoption of a case study approach.

2.5.1.2 Strengths and Weaknesses of the Case Study Research Strategy

From the perspective of this research, the case study research strategy as defined by Robson (1998) and Yin (1994) is considered equivalent to an ethnographic research strategy as defined by Hammersley (1996). This is further explained in section 2.5.2 where the strengths and weaknesses of the ethnographic research strategy are outlined.

2.5.1.3 Case Studies in the Domain of Software Production

Case studies analysing the existing software production situation, rather than devising abstract experiments, are advocated by several researchers. Buckingham-Shum (1996) claimed that a variety of studies show that close study of the design activity "in the natural" is a powerful way to define requirements for subsequent support technology.

Advocates of the case study approach have identified the following problem: commercial barriers can conspire to prevent open research of commercial software production. Confidentiality is one such barrier - with respect to client information, or a software company's development processes. Software development organisations are loath to lay themselves open to analysis. Brooks (1990) suggested that this is one reason that few practitioner case studies have been published. His paper suggests that such case studies have exceptional value and attempts to encourage practitioners to write them and publishers to take them seriously. Axtell *et al.* (1996) also consider there to be a need for detailed case studies.

2.5.2 The Ethnographic Research Strategy

This section begins by defining the term ethnography as a research strategy, before describing the strengths and weaknesses of the strategy.

2.5.2.1 Overview of the Ethnographic Research Strategy

Definition of Ethnography forms part of the semantic spaghetti associated with the definition of the term case study in section 2.5.1.1. Such confusion surrounding the term ethnography (Berg, 1989; Hammersley and Atkinson, 1983; Hammersley, 1996) is likely to have been caused by its diverse roots including anthropology, sociology and psychology. Further confusion is apparent from various well-intentioned authors' attempts to simplify the meaning of certain definitions, often inconsistently. Hammersley (1996) highlights how serious this problem is,

"If what I have said makes it sound as if ethnography is currently in crisis, that is not far from the truth. Most obvious is the crisis of fragmentation: there is no single ethnographic paradigm or community, but a diversity of approaches claiming to be ethnographic (and often disagreeing with each other).” (p15)

‘Participant Observation’ by Spradley (1980) is clearly a book about ethnography but the differing meanings of these terms is not stated, although it is implied that the former is considered a method of the latter. Spradley suggests that ethnography is the “work of describing a culture”(p3) and its central aim is “to understand another way of life from a native’s viewpoint.” (p3). Hammersley and Atkinson (1983) suggest that ethnography is a social science method and describe participant observation as a cognate term. They state that:

“The ethnographer participates, overtly or covertly, in people’s daily lives for an extended period of time, watching what happens, listening to what is said, asking questions; in fact collecting whatever data are available to throw light on the issues with which he or she is concerned.” (p2)

Thus, although Hammersley and Atkinson choose to view ethnography as a method, their definition is broad. However, they suggest that ethnography should not be viewed as an alternative paradigm to quantitative methods.

Berg (1989) uses the “broad umbrella” (p5) term ethnography to cover field research strategies as the term encompasses a “wide combination of elements” such as direct observation, various types of interviewing, listening, document analysis and ethnomethodological experimentation. He also acknowledges that researchers frequently use the term ethnography in different ways and observes that:

“the important point about the concept of ethnography …is that the practice places the researchers in the midst of whatever it is they study. From this vantage, researchers can examine various phenomena as perceived by participants and represent these observations as accounts.” (p52)

Berg justifies treatment of what he calls a “new ethnography” (p53) as a research strategy by comparison with the more traditional textual orientation of ethnography. A further distinction is made between micro- and macro-ethnography. Macro-ethnography aims to describe the entire way of life of a group, whereas micro-ethnography focuses on particular points in time within the group or institution.

Toren (1997) also acknowledges the changes that have taken place in ethnographic analysis. However, she observes that its primary data collection method is still participant observation.

Reconciling his definition of a case study with the definition of ethnography, Robson (1998) describes ethnography as exploratory and finds his definition of case study as broad enough to encompass ethnographic studies. In fact, Robson uses the term “ethnographic case study approach” (p373) to describe his case study definition.

The broad definition of ethnography perhaps renders the term case study redundant. However, there is confusion in the meaning of both terms. For the purposes of this research, both terms are equally valid and will be taken to refer to a broad strategy, rather than a specific method. Hammersley (1996) refers to ethnography as a method but then defines this is such broad terms that it could be regarded as a strategy. The definition is particularly useful:

"In terms of method, generally speaking, the term 'ethnography' refers to social science research that has most of the following features:

(a) People's behaviour is studied in everyday contexts, rather than under experimental conditions created by the research.

(b) Data are gathered from a *range* of sources, but observation and/or relatively informal conversations are usually the main ones.

(c) The approach to data collection is 'unstructured' in the sense that it does not involve following through a detailed plan set up at the beginning; nor are the categories used for interpreting what people say and do pre-given or fixed. This does not mean that the research is unsystematic; simply that initially the data are collected in as raw a form, and on as wide a front, as feasible.

(d) The focus is usually a single setting or group, of relatively small scale. In life history research the focus may even be a single individual.

(e) The analysis of the data involves interpretation of the meanings and functions of human actions and mainly takes the form of verbal descriptions and explanations, with quantification and statistical analysis playing a subordinate role at most." (p2)

Thus, Hammersley's broad definition of ethnography as a method will be treated as a definition of ethnography as a research strategy for the purposes of the current study in order to disassociate the term from its various other meanings.

2.5.2.2 Strengths and Weaknesses of the Ethnographic Research Strategy

There are a number of recognised strengths and weaknesses of conducting ethnographic research and these are outlined below (unless otherwise stated these strengths and weaknesses are largely based on Hammersley, 1996). Strengths and weaknesses due to the nature of the researcher's role on the participant-observation continuum are further explained in section 2.6.1.1.

Strengths

1. Facilitates the development of theory

The ethnographic research process is inductive, thus facilitating exploratory studies rather than being limited to testing explicit hypotheses.

It is thought that preconceptions (or unspoken hypotheses) that the researcher brings with them to an ethnographic study will cease to be maintained in the face of first-hand contact with the people and setting concerned.

2. Flexiblity

Ethnography can proceed without extensive pre-fieldwork design. The strategy and direction of research can change throughout the study in accordance with the focusing suggested by data collection and analysis (e.g. as seen with the funnel technique described in section 2.6.2.3).

Ethnographic research embraces multiple sources of data which can provide the basis for triangulation, reduce the possibility that findings are method dependent (Hammersley and Atkinson, 1983) and address the **reactivity** (see section 2.6.1.1) threat to validity.

3. Studies the social group in its natural setting

Ethnographic research takes a holistic approach to the study of a social situation usually consisting of a group of people in their natural setting. Thus, the risk of 'ecological invalidity' (Hammersley and Atkinson, 1983) of the study is far less of a risk than it is with experimentation or survey research strategies. However, it is important to note that reactive effects (also known as the 'Hawthorne effect') can pose a threat to the validity of an ethnographic study (validity is covered in greater detail in section 2.7.4).

Weaknesses

1. Limited potential for generalisable findings

Ethnographic research usually sacrifices the study of a breadth of cases in favour of studying a small number of cases in depth. Often ethnographers are not concerned with empirical generalisation but rather with making theoretical inferences for which cases studied do not need to be representative (Hammersley, 1996).

2. Findings are based upon the 'human instrument'

Ethnographic research is dependent on the data collection, interpretation and analysis of the researcher as a human instrument (Hammersley, 1996; Yin, 1994; Robson, 1998). This is not necessarily a weakness of ethnographic research as long as this aspect of the research is understood and accounted for. Various techniques, such as triangulation may be employed to ensure that a researcher's findings are not idiosyncratic.

3. Large volumes of data

Ethnographic research usually faces the practical problem of recording, organising and analysing large volumes of qualitative data.

4. Practicalities of longitudinal studies

Ethnographic research often requires the researcher to become a participant-observer of a group of people being studied over long periods. This can impose real constraints on ethnographic studies and can be demanding for the researcher.

5. Ethical problems of 'complete participant' studies

Difficult ethical questions can arise when a researcher undertakes covert observation - the usual form of the complete-participant role. However, such a role is often desirable in order to gain access to a setting or eliminate reactive effects.

2.5.3 Other Qualitative Methodologies

A number of other qualitative methodologies were assessed in relation to this research. These methodologies were considered to be inappropriate but are covered here for completeness. In fact, it is questionable whether the following constitute complete methodologies at all for the purposes of most studies instead they seem to provide techniques which can be deployed in a range of qualitative research. Or, in the case of grounded theory, an underlying principle is introduced which is desirable for much qualitative research.

2.5.3.1 Content Analysis

Content analysis is an approach to the study of the entire range of communicative and symbolic media, including verbal dialogues, films, advertisements, cartoons, theatre and political speeches (Krippendorf, 1980). From these forms of data, researchers make inferences about the subject of interest.

Content analysis techniques aim to break the selected data into information-giving units. Sampling is then utilised to select a representative set of these units. Finally, units are organised into coding categories. Further analysis is usually concerned with organising the categories into meaningful patterns and making logical links between them (Henwood, 1997).

Content analysis usually attempts to derive inference from existing material. Therefore, from the perspective of an exploratory study it must be viewed as a technique to be drawn on rather than a particular research strategy.

2.5.3.2 Protocol Analysis

Protocol analysis is very similar to content analysis (Henwood, 1997). The difference is that the data consists of verbal protocols. Typically, verbal protocols are generated when an individual is asked to 'think aloud' during the completion of a cognitive task (Gilhooly and Green, 1997). This technique is often used in knowledge elicitation for knowledge-based systems where software designers are trying to capture a human expert's knowledge.

From the perspective of this research, protocol analysis is also viewed as a tool rather than a research strategy.

2.5.3.3 Grounded Theory

Grounded theory specifies that new developments in theory can be made by close and detailed inspection of particular problem domains or settings (Henwood, 1997). Thus, this methodology aims for the discovery of new theories that are well grounded in the data. The concept of generating theory from field-work or case study documentation is an important principle of contemporary qualitative research (Pidgeon, 1997). Richards and Richards (1991) also consider grounded theory as desirable when making theory from data but do not consider it a method for handling data. Bryman and Burgess (1994) conclude that there are two main influences of grounded theory. Firstly, it has influenced contemporary qualitative research such that there is now a general desirability of extracting concepts and theory out of data. For example, Spradley (1980) states that ethnography is an excellent strategy for the discovery of grounded theory. Secondly, they suggest that grounded theory has informed qualitative data analysis, particularly the introduction of coding for concept creation.

A key defining feature of grounded theory is that it attempts to declare an explicit method for many usually implicit aspects of qualitative research (Henwood, 1997). This method has similarities to content analysis, which are illustrated by this summary:

"After some data collection and reflection in relation to a general issue of concern, the researcher generates categories which fit the data. Further research is undertaken until the categories are 'saturated', that is, the researcher feels assured about their meaning and importance. The researcher then attempts to formulate more general (and possibly more abstract) expressions of these categories… This stage may spur the researcher to further theoretical reflection and in particular he or she should by now be concerned with the interconnections among categories…" (Bryman & Burgess, 1994b; p4)

Although this method exists it is questionable whether it is ever actually used in the context of an overall ground theory research strategy. For example, Bryman and Burgess (1994) conclude that it is rare to find evidence of "the iterative interplay of data collection and analysis that lies at the heart of ground theory" (p221), and rarely have they found clear indications that theory is being developed. In fact, Burgess (1984) questions what 'theory' actually means in this sense, uncovering the suggestion that it refers to properties, categories and hypotheses.

There is evidence to suggest that qualitative studies often pay lip service to grounded theory, whilst disguising the precise process of grounded theory analysis undertaken (Bryman and Burgess, 1994). Richards and Richards (1991) highlight this when commenting that grounded theory "is widely adopted as a bumper sticker in qualitative studies" (p43).

For the purposes of this research, grounded theory will be viewed in two ways. Firstly as an underlying principle of the research, i.e. theory should emerge from the data. Secondly, the analysis techniques of grounded theory (e.g. the use of raw data coding leading to concept creation) will be used to inform qualitative analysis. For the current study, grounded theory is not regarded as a complete research strategy as it appears to be rarely used in this way.

2.5.4 Summary and Final Commentary

This investigation has concluded that the terms ‘case study’ and ‘ethnography’ are not clearly and consistently defined. However, the most appropriate definitions of both are very broad and suggest that the terms relate to a flexible strategy for conducting qualitative research. Case study is a strategy for studying particular cases; ethnography is also case-based. Although ethnographic studies could almost always be labelled as case studies, the converse is not necessarily true.

The research question requires the assessment of the effect of introducing an HCI designer role into a commercial software project (section 2.2). This suggests that an exploratory approach would need to be taken to studying a particular case. However, the characteristics of the research design highlight the problem of finding an appropriate case to study. A broad case study strategy may suit a situation where a number of possible cases are available for study. However, with no existing cases to study, this research is directed to an ethnographic strategy that selects an appropriate software project and installs the researcher as an HCI designer and as a participant-observer. In this way, a case can be created for the study. This participant-observation emphasis of the study directs the qualitative investigation to an ethnographic strategy.

The next section evaluates the methods that are available within the context of an ethnographic strategy.

2.6 Methods available within an Ethnographic Strategy

Following the selection of an ethnographic research strategy (see section 2.5.4), this section focuses on the methods of ethnographic research, particularly those relating to data collections and analysis

Historically, it seems that advice given to ethnographers has been to 'just go and do it' (Hammersley and Atkinson, 1983). The majority of researchers still acknowledge that there are no universal procedures for doing ethnographic research, for example Rachel (1997) describes it as a craft and Robson (1998) suggests that many consider it more of an art than a science. However, some of the fundamentals of the ethnographic strategy do appear to be consistent.

Following their review of qualitative data analysis, Bryman and Burgess (1994) state that data collection and analysis are not considered distinct phases of qualitative research (Hammersley and Atkinson (1983) also share this view), because ideas develop inductively and are then compared against more data (Schutt, 1996; Robson 1998). Burgess (1984) suggests that research design will be continually modified and developed by the researcher and sums up with the following:

"Doing field research is, therefore, not merely the use of a set of uniform techniques but depends on a complex interaction between the research problem, the researcher and those who are researched. It is on this basis that the researcher is an active decision maker who decides on the most appropriate conceptual and methodological tools that can be used to collect and analyse the data."(p6)

However, although there is a fundamental intertwining of data collection and analysis, it is possible to describe them distinctly (if somewhat simplistically).

2.6.1 Ethnographic Data Collection Methods

There are three primary methods of ethnographic data collection: participant observation, interviewing and document analysis (Burgess, 1984; Hammersley and Atkinson, 1983) and these methods are often utilised within the same ethnographic study. It is this openness to diverse data sources that is considered a particular strength of what Yin (1994) and Robson (1998) describe as the case study research strategy which is very similar to the definition of an ethnographic research strategy used here.

Robson (1998) asserts that there is no general best method of data collection, rather the methods chosen should be driven by the kind of research questions asked, and moderated by what is feasible in terms of time, resources, skills and expertise.

2.6.1.1 Participant-Observation

The term participant observation refers to a continuum of roles and it is necessary for the researcher to decide what degree of participation or observation is appropriate (Schutt, 1996). Because participant observation is the primary method of data collection for an ethnographic strategy (Toren, 1997), decisions relating to this method are an important preliminary consideration.

Although it is usual for a researcher to adopt a role between the extremes of complete participant and complete observer, the extremes are discussed below for the benefit of illustration.

The complete observer attempts to see things as they happen without causing any disruption to the situation studied. However, what actually happens in this situation is that the observer sees what happens in a situation when it is being observed. So, the Hawthorne effect is a feature of this research role, also known as **reactive effects** (Schutt, 1996). The social setting for the research is thought to have some bearing on the reactive effects. For example, if the complete observer is in a setting with a large number of other people such that they do not attract particular attention, it is likely that reactive effects will be low. However, if they are in a situation with few people and the act of observing is obviously unusual, then reactive effects are likely to be of greater concern.

The complete participant role can be used to gain access to otherwise inaccessible settings or to lessen reactive effects (Schutt, 1996). The complete participant role often takes the form of covert observation. This kind of covert research raises a number of problems, some relating to the researcher's need to hide their identity and behave as normal group members. For example, note taking and questioning group members is troublesome. Ethical issues are also a primary consideration in covert observation as it is not possible to foresee the consequences of the researcher's involvement with the group being studied.

It is usual for researchers to adopt a role in between complete observer and complete participant. Most participant observers who disclose the nature of their research activities to the group being studied, report that after they have become known and trusted, they don't believe their actions have any tangible effect on members' actions (Schutt, 1996).

Hammersley and Atkinson (1983) attack the attempts of rigid positivistic or naturalistic approaches to eliminate the effects of the researcher on the data. They instead suggest it is essential that the reflexive character of social research is emphasised. This they describe as recognition that "we are part of the social world we study. … This is not a matter for methodological commitment, it is an existential fact."(p14). Similarly, Barley (1986) states that it is not possible to get away from the fact that the study affects the observed. Hammersley and Atkinson (1983) therefore highlight the need to examine the relationship that the researcher has to the situation being studied and the effects this causes. The need to focus attention on the ethnographer and the effect they have on the research is becoming generally recognised as desirable (Spradley, 1980; Schutt, 1996; Rachel, 1997; Toren, 1997; Robson, 1998; Burgess, 1984).

The need for the researcher to consider themselves a research instrument is often highlighted (Robson, 1998; Spradley, 1980). Considered as an instrument, the researcher is encouraged to become introspective in their ethnographic accounts. The impression an ethnographer gives to the group being studied by virtue of their appearance, speech and behaviour are another important consideration of the 'human instrument' (Hammersley and Atkinson, 1983). In order to gain acceptance within a group the ethnographer must be aware of the importance of such issues. Schutt (1996) suggests that matching a field researcher's social attributes (age, sex, race, etc.) with those studied can be advantageous in some projects. Robson (1998) also supports the view that the researcher should have some familiarity with the phenomenon and setting under study.

Note taking is an aspect of participant observation that needs special attention as it is the primary means of recording participant observation (Schutt, 1996). Hammersley and Atkinson (1983) suggest that it is often not possible to make notes during participant observation, particularly if the researcher is in the role of full participant. They state that note taking must be "congruent with the context of the setting under scrutiny" (p147). This is because continuous note taking can be conspicuous and may appear threatening or inappropriate. Thus, taking extensive notes whilst in the field is often considered too disruptive so researchers are advised to make brief notes throughout the day to serve as memory joggers for a full write-up within 24 hours (Robson, 1998; Schutt, 1996).

Some researchers suggest that it is impossible to ever record all data acquired in the course of fieldwork, especially when the participant observer is in the field for a long period of time (Hammersley and Atkinson, 1983; Barley, 1986). Therefore, there is a need for note taking to be somewhat selective. The note taking during the early stages of a study will be necessarily general, and this will focus as emergent issues arise (Hammersley and Atkinson, 1983). They suggest that as theoretical ideas develop what is significant changes and this is reflected in a changing emphasis in the field notes. Thus, field notes also become an essential aspect of ethnographic analysis. Ethnographic field notes will often suggest new concepts, causal connections and theoretical propositions (Schutt, 1996). Schutt also suggests that the analysis of field notes typically proceeds sequentially, with the researcher first identifying problems and concepts that appear useful in understanding the situation. Then, as observation and reflection continues, these problems and concepts are refined.

2.6.1.2 Ethnographic Interviewing

Asking questions forms part of most participant observation and sometimes this takes the form of interviewing (Schutt, 1996). Spradley (1980) draws a distinction between informal and formal interviewing. The former occurs whenever the ethnographer asks someone a question during the course of a participant observation and the latter is a formal interview situation (following a request). The terminology used to describe Spradley's 'formal interviewing' is inconsistent, for example, Schutt describes it as ‘intensive interviewing’, Hammersley and Atkinson (1983) refer to it as ‘reflexive interviewing’ and Burgess (1984) describes it as ‘unstructured interviewing’. Hammersley and Atkinson oppose the unstructured/structured split sometimes attributed to the difference between ethnographic interviewing and survey interviewing. Instead, they prefer a ‘standardised’ versus ‘reflexive’ distinction. With ‘reflexive interviews’ or ‘intensive interviews’, the ethnographer does not decide beforehand on the questions that will be asked, rather they have a list of issues or topics to be covered (Hammersley and Atkinson, 1983; Schutt, 1996).

One weakness of formal ethnographic interviews is that they are one step removed from the natural social context. However, the strength of the formal interviews is that they enable the researcher to follow up observations in the social context with particular individuals more intensively than the context may otherwise permit. Informal interviews share this strength without the disadvantages associated with moving away from the social context.

2.6.1.3 Document Analysis

A part of the rationale for early ethnography came from the fact that many early peoples studied by this research strategy had not previously had a written history (Hammersley and Atkinson, 1983). Because ethnography now commonly occurs in a literate setting, it has become relevant to incorporate some analysis of documentation within such research. An important methodological difference between document analysis and other ethnographic methods is that instead of directly observing for the purposes of enquiry, with documents it is necessary to deal with something that was produced for some other purpose (Robson, 1998). Yin (1994) suggests that documentary evidence is relevant to almost every form of case study (taken to mean ethnographic case study from the earlier discussion of terminology). He finds that the most important use of documents is to corroborate and augment evidence from other sources. Yin suggests that the collation of documentation is an important aspect of any data collection plan.

The strengths of document analysis include the corroborative potential this represents as a form of alternative evidence; and reduced (or eliminated) reactive effects because documents are produced without any involvement of the researcher (either as participant or observer). A general weakness of documentary evidence stems from the difficulty of interpreting it in context and the means by which it is analysed. Similarly, it is likely that document analysis will have to contend with a large volume data.

More typical ethnographic document analysis is likely to take the form of analysing personal documents, such as diaries, autobiographies or life histories (Burgess, 1984). Documentation of this form suffers a number of weaknesses including authenticity (i.e. the need to be aware of potential forgery or misinterpretation); distortion and deception (i.e. exaggeration and misrepresentation); and problems of sampling (i.e. how documents are selected to be representative of a particular situation).

2.6.2 Ethnographic Analysis Methods

That some people consider ethnographic research a craft skill (e.g. Rachel, 1997) is particularly apparent from texts describing approaches to the analysis of ethnographic data. Bryman and Burgess (1994) attempted to confront this issue head-on with their book, "Analysing Qualitative Data", where they encouraged contributors to describe the techniques employed during their qualitative analysis. They attempted this because they found qualitative texts frequently avoided the issue and published qualitative work usually failed to describe analysis methods used. For the reviewer of qualitative analysis techniques, it is highly disappointing that after preparing a collection of papers on the subject, Bryman and Burgess are forced to conclude:

"…it is still not absolutely clear how issues or ideas emerge in order to end up with the finished product. …the real problem is that we simply do not know why certain themes emerge as core elements in the report…" (p224)

There are a number techniques/methods/approaches to analysing qualitative data, although they are often not precisely prescriptive. As Robson (1998) suggests there is much emphasis on the quality of the analyst and interpretation when qualitative data is involved and this precludes reducing the task to a defined formula.

This section begins by introducing chronological analysis. Triangulation is then described as a versatile technique that may be utilised within other aspects of analysis (particularly those described later in this section). The 'funnel approach', which demonstrates the intertwining of ethnographic data collection and analysis, is then introduced. This approach highlights the importance of **coding** and **conceptualisation** of data as part of the analysis phase. These aspects of analysis are main elements from the conclusion of Bryman and Burgess' work on qualitative data analysis (1994) and are utilised as headings below.

2.6.2.1 Chronologies

A chronological analysis can be appropriate for longitudinal studies. One form of such analysis is the life history approach – usually meaning a detailed account of a single person's life. Robson (1998) suggests that:

"'mini life histories' …covering in some detail one or more individuals' involvement with, say, an intervention which forms the basis of a case study …can be a useful component of analysis" (p382).

2.6.2.2 Triangulation

Robson (1998) describes triangulation as "an indispensable tool in real world enquiry". However, he believes the scope of triangulation is broad, ranging from the use of multiple methods to gaining information on a topic from several informants. Fundamentally, triangulation involves testing one source of information or interpretation against other sources. Stake (1995) describes several triangulation protocols including 'data source triangulation', 'investigator triangulation' and 'theory triangulation'. 'Data source triangulation' aims to check whether what is being observed and reported carries the same meaning under different circumstances. Perhaps another definition of this would come from the utilisation of multiple sources of evidence relating to the same phenomena, which would conform to Yin's (1994) description of triangulation (which Yin uses to address issues of construct validity). 'Investigator triangulation' aims to triangulate the interpretation of the phenomena under investigation by allowing other researchers to analyse the data. Finally, 'theory triangulation' aims to triangulate the final description to assess the extent to which reviewers agree on the meaning or interpretation of the analysis. Fundamentally, triangulation is an excellent technique for improving the validity of analysis and interpretations. Therefore, triangulation clearly has a place in the coding and conceptualisation stages of qualitative data analysis.

2.6.2.3 Focusing the analysis - the 'Funnel'

A number of researchers utilise a **funnel** analogy to demonstrate the progressive focusing of ethnographic research (Hammersley and Atkinson, 1983; Spradley, 1980). This is used to describe the inductive progression from what is often an exploratory research question (Schutt, 1996; Robson, 1998), through the formulation of general concepts to make sense of specific observations (Schutt, 1996). Spradley (1980) describes the broad rim of the funnel as consisting of *descriptive observations* aiming to catch everything that is going on. He suggests that the funnel narrows as *focused observations* are made, requiring the scope of the research to narrow by focusing on particular categories. At the bottom of Spradley's funnel is the narrow opening representing *selective observations* – the smallest focus for observations that will be made. This funnel is a core component of Spradley's (1980) Developmental Research Sequence (D.R.S.) method for conducting ethnographic research. Clearly, the funnel describes an aspect of data collection as well as analysis but serves to illustrate the means by which an ethnographic study can gradually focus.

2.6.2.4 Coding the Data

Coding is an essential process in the analysis of qualitative data as it organises the volume of notes collected and represents the first stage in conceptualisation (Bryman and Burgess, 1994). From their review of qualitative analysis, Bryman and Burgess conclude that there is considerable confusion over what the term 'coding' means. However, regardless of the precise procedure adopted it is clear that some form of coding takes place as part of most ethnographic studies. The influence of grounded theory has clearly contributed to this analytical process (c.f. section 2.5.3.3).

Robson (1998) defines a code as a symbol applied to a group of words to classify or categorise them. He suggests that codes are retrieval and organising devices which aid the gathering together of all instances of a particular kind. He further suggests that this is **essentially the same as developing a category system in content analysis**. Miles and Huberman (1984) have defined two levels of coding. First level coding is concerned with attaching labels to groups of words and second level coding consists of grouping the initial codes into fewer patterns or themes. They suggest this is the qualitative data analysis equivalent to factor or cluster analysis of quantitative data.

The following guidelines for coding qualitative data are adapted from Robson (1998):

1. Discover and code (provisionally name) categories in the data.
2. Relate categories to the contexts in which they occur.
3. Relate categories to each other; construct sub-categories where appropriate.
4. Base the categories on specific data.
5. Develop core categories, relating all categories and sub-categories to the core.
6. Discard totally or largely unrelated categories.

Robson's guidelines begin with what some researchers class as 'coding' and then evolve into what other researchers would describe as the early stages of conceptualisation. The second level coding of Miles and Huberman (1984) described above would also be regarded as the early stages of conceptualisation by some. For example, Mason (1994) describes an approach she took to the analysis of a large qualitative data set. Firstly, data were searched for themes and developed into 35 descriptive analytic categories – this corresponds to what has been described as 'coding' above. Mason is clear that even this stage involved interpretation. The second phase is described as much more involved and is described as "teasing out" conceptual categories and the relationships between them based on theoretical perspectives they brought to the study and from a grounding in the data itself. Ultimately, this generated too many conceptual categories, resulting in a process of subdivision and amalgamation of categories at a later stage. This second phase could also be described as conceptualising the data (see section 2.6.2.5).

In summary, coding proceeds by the tentative development and labelling of those concepts in the text that the researcher considers to be of potential relevance to the problem being studied. That this is a complex procedure is clear from the fact that judgement is always involved in this labelling process.

2.6.2.5 Conceptualising the Data

There appears to be a continuum between coding and conceptualisation as described by Bryman and Burgess (1994) and others. Coding is said to involve the early stages of conceptualisation and conceptualisation is described as an extension of coding. Clearly, the researcher is applying their own interpretation to ethnographic field notes in order to generate both coding and conceptualisation. **The aim of conceptualisation is the generation of concepts that can be used in the formulation of theory** (Bryman and Burgess, 1994).

Turner (1994) describes a conceptualisation approach akin to card sorting from the domain of knowledge elicitation. After labelling the concepts (by working paragraph by paragraph through a report) he recorded each concept label on a piece of card. These cards were then "sifted, sorted and juggled into a coherent theoretical model." (p198). The next step involves a search for causal and other links whilst moving towards the development of a **theoretical pattern**.

Ritchie and Spencer (1994) describe a process of abstraction and conceptualisation of data. They suggest that the analyst attempts to identify the key issues, concepts and themes in the field data, ultimately setting up a **thematic framework** within which to facilitate **conceptualisation** of the data. They also claim that the development of this framework requires interpretation and intuition.

Having developed the thematic framework and used this for the preliminary analysis, Ritchie and Spencer (1994) suggest that the analyst turns to 'charting' in order to build up a picture of the data as a whole. They describe charts as headings or sub-headings drawn from the thematic framework. After this 'charting', the analyst pulls together key characteristics of the data to map and interpret the data as a whole. Mason (1994) and Robson (1998) also cite the usefulness of visual representations (including layouts, plans, maps and diagrams). Robson suggests that a visual representation can facilitate the formulation of **an abstract representation of the data** on a single sheet of paper.

Ritchie and Spencer (1994) describe the realities of the analysis leading to the production of mapping and interpretation as follows:

"this part of the analytical process is the most difficult to describe. Any representation appears to suggest that the analyst works in a mechanical way, making obvious conceptualisations and connections, whereas in reality each step requires leaps of intuition and imagination" (p186)

Clearly, this is another dimension of qualitative data analysis for which there is no specific prescribed procedure.

Returning to the conclusions of Bryman and Burgess' (1994) review of qualitative analysis, they found that their contributors had provided insight into conceptualisation but had been more guarded about the emergence of theory. They point out that although concepts are the building blocks of theories, they are not theories in themselves. Similarly, they could not draw a firm link to specific theory from the relationships described between concepts by several of their contributors. They finally conclude that although there is frequent mention of grounded theory, there is a lack of certainty about the degree to which theory is being generated from the process of conceptualisation.

2.7 Further Methodological Considerations

Section 2.6 covered data collection and analysis methods within an ethnographic strategy. This section covers further methodological considerations associated with ethnographic research, because there are a number of other practical considerations that an ethnographic study must address. In terms of the current study, further methodological considerations include: entering the field, developing and maintaining appropriate relationships with people in the setting, sampling, and validity.

2.7.1 Entering the field

The fundamental importance of the researcher as the human instrument must be acknowledged from the very start of an ethnographic study. As with a typical anthropological study, a certain degree of background work is necessary before entering the field. Lincoln and Guba (1985) warn that "one would not expect individuals to function adequately as human instruments without an extensive background of training and exposure." (p195). However, some researchers are against systematic study of the proposed setting for fear that it will introduce bias (Schutt, 1996).

As a minimum requirement Schutt advises that researchers' learn how participants dress and what their typical activities are. This advice is intended to prepare the researcher for the general ethnographic approach and would be especially appropriate for the street-corner studies frequently cited by Schutt (1996). However, the intent of this advice is of great importance for an ethnographer as it relates to the need to gain acceptance and establish rapport with the group to be studied. Schutt also advises of the need to be sensitive to first impressions made and ties established during the fieldwork, which would seem to be relevant to all forms of ethnography.

Another aspect of preparing for ethnographic research is the need to learn the language spoken by the group to be studied. Gaining some language skills relevant to the group is an important aspect of many traditional anthropological studies. As with studies of remote tribes, language skills gained in preparation for an ethnographic study will be built upon throughout the study in order that the researcher can properly understand what is being observed.

Entering the field also requires the negotiation of access to the group and setting to be studied and this can be quite a problem (Hammersley and Atkinson, 1983). Robson (1998) provides a great deal of practical advice on discovering an appropriate group or setting and negotiating access. In particular, he outlines the necessarily opportunistic nature of gaining openings for field research and the need to be flexible.

Schutt (1996) highlights the need to take a flexible approach to entering the field, and cites the example of Liebow's seminal Tally's Corner study to illustrate the point. He describes how Liebow discarded his initial intention of studying individuals on several city blocks when he became deeply involved with the first group he encountered.

2.7.2 Developing and Maintaining Relationships

Because most ethnographic studies are carried out over a period of time, it is essential that the researcher develops and maintains relationships with members of the group under study (Robson, 1998; Schutt, 1996; Burgess, 1984). The group must feel that the researcher can be trusted and, in particular, relied upon to keep promises of confidentiality (Robson, 1998; Schutt, 1996). Establishing trust is essential to enable the researcher to blend into the situation and observe people in their natural setting (Burgess, 1984), thereby reducing reactive effects.

On a practical level, Schutt (1996) recommends several guidelines, which include: develop a plausible (and honest) explanation for yourself and your research; maintain support of key individuals in groups or organisations under study; do not be too aggressive in questioning others.

2.7.3 Sampling

Sampling is no less important for ethnographic research than any other kind of research (Schutt, 1996). Typically ethnographic research focuses on an in-depth study of a particular case or a small set of cases, and this always calls into question the general relevance of the findings outside the context of the study (Hammersley and Atkinson, 1983). However, it can be argued that an in-depth study of a particular case is more revealing and produces more generalisable findings than a survey approach covering a broad array of cases in little depth. Schutt (1996) cited Liebow's 'Tally's Corner' in-depth study of a particular group as an example of a single-setting study believed to be representative of similar groups. However, Schutt also suggests that the findings could have been strengthened by a comparative study of similar groups by means of theoretical sampling.

Hammersley and Atkinson (1983) suggest that the stage of development of relevant theory in a particular area will direct the strategy used to select cases. They suggest that for early phases of the generation of theory, the cases chosen for investigation may not matter greatly. Only later in the development and testing of theory does the selection of cases take on particular importance.

Robson (1998) observes that often 'real life' is a factor in sampling, directing various constraints such as the degree of access to a setting, availability of people within the setting and other logistical considerations. He also advises on the need to consider sampling in relation to the research questions.

2.7.4 Validity

Robson (1998) contends that the concepts of internal validity, external validity, reliability and objectivity have been developed alongside experimental and survey research and are not appropriate to qualitative data. Instead, he describes related concepts of credibility, transferability, dependability and confirmability. The more traditional quantitative headings are followed below, but the discussion is allied to the concepts described by Robson. Construct validity in the context of ethnographic research was not covered by Robson's concepts, but this is thought to have some relevance and is therefore described.

2.7.4.1 Construct Validity

Construct validity is often typified by the question, 'have I measured what I think I've measured?' Robson (1998) suggests that the complexities of answering this question can lead to "an unhealthy concentration on this aspect of carrying out an enquiry". He contends that for many studies there is an intuitive reasonableness to assertions that a certain approach provides an appropriate measure.

One method of dealing with issues of construct validity is to take a multi-method approach to data collection (Robson, 1998). However, when it comes to construct validity of qualitative data, Robson takes a step backwards and does not confront the issue. Yin (1994) suggests that case study researchers can address the issue in three ways: using multiple sources of evidence; by establishing a chain of evidence during data collection; and finally the review of draft case study reports by key informants.

Hammersley and Atkinson (1983) view construct validity as taking on a different form in ethnography. This is because of the interplay between finding indicators and conceptualising analytic categories – the inductive nature of ethnography. However, they conclude that identifying standard indicators is inappropriate to ethnographic research and is not an essential feature of theory.

2.7.4.2 Internal Validity

Robson (1998) describes this as 'credibility' and its goal as the demonstration of an accurate portrayal of the subject of enquiry. Within an ethnographic study, the greatest threat to internal validity comes from the 'human instrument' and their inferences. **Reactive effects** of study participants to the presence of the researcher may also pose a threat to internal validity. Robson proposes a number of means to address concerns about internal validity: prolonged involvement of the researcher in the field; persistent observation of specific situations within the study; triangulation of evidence from different sources, different methods or different investigators; peer debriefing on a continuous basis.

2.7.4.3 External Validity

External validity refers to the extent to which findings are generalisable; Robson (1998) describes this as 'transferability'. He also suggests that claims of generalisation to a population are often inappropriate following the study of a single case. Yin (1994) suggests that the notion of generalising to other cases is wrong and instead suggests that the aim should be to generalise findings to theory. Determining the general applicability of findings from an in-depth ethnographic study is often the subject of further study.

2.7.4.4 Reliability

Robson (1998) defines reliability as 'dependability' and goes on to suggest that a kind of audit trail should be left by the researcher so that others may check the processes followed. The goal of reliability considerations is to minimise errors and bias in the study. The general approach suggested is to document procedures followed in sufficient detail for another researcher to repeat the study and arrive at the same results (Yin, 1994).

2.8 Methodology in Context

The ethnographic nature of this research dictates the need to describe the methodology as it evolved in the context of the study, rather than as a pre-defined procedure. This is due to the evolving nature of ethnographic research as a study progresses, defined in section 2.6.2.3 as the ‘funnel’, and fuelled by the intertwining of the data collection and analysis stages with ethnographic research (see section 2.6). Thus, it has been necessary to provide some outline of the results in describing the methodology in order to be clear about the analytical process followed.

This section also introduces some ethnographic-style descriptions written in the first person to show where inferences and analysis have come from interpretation of a situation and to aid clarity in general (this is in accordance with recommendations of the Human Factors Society, 1992).

2.8.1 Overview – Selection of an Ethnographic Approach

The research question (section 2.2) and required characteristics of the research design (section 2.3) pointed to the need for an exploratory study (section 2.4). It was concluded that a longitudinal qualitative investigation would provide the best means of answering the research question. An ethnographic approach (taking the definition of ethnography as a research strategy, also encompassing the definition of case study research) was thought to be the most appropriate qualitative methodology for undertaking this longitudinal investigation.

The **strengths** of the ethnographic approach (see section 2.5.2.2) map closely to the research question and its surrounding characteristics:

- The research question is necessarily exploratory due to the lack of research on: the diverse potential of prototypes within a software team; the concept of conceptual integrity; the idea of an HCI designer as a full software team member. Thus, the inductive nature of ethnographic research facilitates the development of theory from such exploratory research.

- The research question aimed to analyse the effect of introducing an HCI designer into a commercial software team. This study is without precedent so extensive pre-fieldwork design was not considered viable. The ethnographic approach provided a flexible framework ideally suited to this kind of study. In particular, the approach opened the door to the study of a full-time HCI designer in a commercial software team by installing the researcher (as a participant-observer) into an existing team to perform this role. This situation was sufficiently rare that this was considered almost the only way to perform such a study.

- There is a growing body of evidence that HCI theory is often of little use or relevance to a practical software development situation (see section 1.9). Therefore, the ethnographic approach provided a highly appropriate means of ensuring the practical relevance of the research.

Further practical reasons dictated the selection of an ethnographic research strategy. The research opportunity presented to the researcher was to carry out HCI research at Cranfield University whilst working full-time for an Independent Research Organisation (IRO) as an HCI designer (this formed part of a new Postgraduate Training Partnership (PTP) scheme). Thus, the researcher had commercial obligations to the IRO as well as research motivations. An ethnographic strategy provided the perfect means to exploit this situation. The PTP scheme had clearly taken care of several typical difficulties of ethnographic research including selecting an appropriate setting to study (see 'Sampling' section 2.7.3) and gaining access to the field (see 'Entering the Field' section 2.7.1).

2.8.2 Beginning the Participant-Observation

The researcher began working as an HCI designer for SFK Technology (the software subsidiary of British Hydromechanics Research Group (BHRG)) in September 1992. Having arranged the setting to study, the next major decisions related to the participant-observation method of data collection. Participant-observation is the primary method of data collection for ethnographic research and it was clear from the outset that the researcher was going to have to 'pay their way' by being the HCI designer for SFK. Therefore, it was evident that the participant-observation method of analysis would be the most appropriate way to proceed with the research in this situation. In fact, the commercial conditions dictated to some extent the role of the researcher along the participant – observer continuum of roles. Essentially, the researcher was to be a complete participant in the setting.

The PTP scheme provided a useful cover for complete participation without the need for it to be covert research as complete participant studies often are. For example, presentations about the research were made to the whole company. However, I firmly believe that most people believed the research to be part-time university study on my part, rather than a study specifically involving them. The openness about the research activity removed the difficult ethical problem associated with covert research (see section 2.6.1.1). My work as an HCI designer within the software project was seen by most people in the setting as 'paying my dues' in respect of the support from my employer (SFK) for my 'part-time' PhD at Cranfield University. The reality of the situation was that I had a full-time job to perform and a full-time research commitment – not an uncommon scenario in ethnographic research. This perception people had of me as a part-time student supported by my employer provided an excellent cover story for participation in SFK's main software team and my research intentions. Although everybody was well aware of my research, this was soon forgotten amid the day-to-day traumas of software production. Therefore, possible reactive effects from other participants in the setting to the presence of a researcher were considered extremely unlikely – I was perceived as an HCI designer and not a researcher. The complete participant role is often considered to be a means of reducing reactive effects (see section 2.6.1.1) and my experience from this ethnographic study strongly supports these claims.

The social attributes of the researcher matched very well to what other participants would regard as ‘normal’ for a software professional - appropriate first degrees, experience of software development, age, sex, race and upbringing. Thus, the researcher was well matched to the regular participants on social attributes. The social attributes of the researcher are often considered an important aid to gaining acceptance within the group studied (see section 2.6.1.1). The fact that I ‘started work’ as an employee responsible for HCI design (as fitted my background and experience) made my cover story even more credible. Furthermore, developing and maintaining relationships with members of the group studied was achieved in a similar fashion to any new employee joining a company. Maintaining regular contact with a research supervisor (at Cranfield University) throughout the study was an essential means of continuing to maintain a research focus to my activities at SFK.

Note taking is a vital aspect of ethnographic research and it is often suggested that brief notes should be made throughout the day where possible, if this can be done without appearing conspicuous and inducing reactive effects. Fortunately, all software engineers at SFK were encouraged to jot down notes in hard-bound A4 log books. It was completely normal for software engineers to walk around carrying such notebooks (particularly into meetings), so my grey notebook and occasional note-taking was nothing different from the norm (though I was perhaps perceived as particularly diligent). The use of video or audio recording devices was strictly avoided, as this would have very quickly altered the team’s perception of my position from ‘the HCI designer’ to ‘researcher’. Reactive effects would have perhaps even spiralled from this heightened awareness of my research role.

2.8.3 The Human Instrument – The Researcher

The human instrument is possibly the most important methodological consideration in an ethnographic study. The skills of the researcher and their ability to make appropriate interpretations are essential to considerations of internal validity (see section 2.7.4.2). It is apparent that the researcher needs to have certain skills, capability to understand (or learn) the language of the group being studied and is an appropriate ‘fit’ to the group being studied (see section 2.6.1.1).

It is therefore relevant to consider various elements of the ethnographic researcher’s skills and experience, which are summarised in table 2.1.

Table 2.1 Summary of researcher's skills and abilities

Skills	Experience	Effect on study
Software	- Programming as a hobby for 15 years - BSc and MSc in software related degrees - 18 months previous experience working in software teams	- The software skills of the researcher facilitated comprehension of complex technical issues in the software development - The researcher had a good understanding of the 'language' used by software professionals and so was readily able to communicate with the group studied
Research	- Research experience was that introduced during BSc and MSc courses. - During the early stages of this research, a number of relevant courses on Cranfield University's Occupational Psychology MSc we taken	- The researcher had received appropriate research methods training
Study of organisations, groups, individuals and communication	- No experience	- As the researcher had little previous exposure to studies of organisations, groups, individuals or communication he had no preconceived ideas along these lines
HCI design	- University HCI courses at Middlesex Polytechnic and Cranfield University	- A potential area of study bias was the inexperience of the researcher as an HCI practitioner and unfamiliarity with the company and team

Perhaps the most important aspect of my skills was my long experience of software and programming. Kim (1990) contends that a programmer will not trust you unless you have written a program and experienced the basic drama of so doing (see section 1.10.2). Whether this is true or not, it is clear that having a familiarity with the 'language' of software engineers and some understanding of the roots of the jargon (e.g. software engineering history and concepts) enabled me to understand what software engineers were talking about.

That I had no practical experience of carrying out studies of this kind was a potential weakness of the study. This was addressed at the outset of the study, by taking a selection of courses from Cranfield University's MSc Applied Psychology course.

One potential area of bias was due to my lack of experience of the 'HCI designer' role (this bias was unavoidable as such a role was considered almost unheard of). This potential bias was addressed by the longitudinal nature of the investigation. Ultimately, the bias was also addressed by extending the research to cover a further study with almost the same team, after having performed the HCI role for a year.

2.8.4 Other Human Instruments Involved – Peer Interpretation/ Triangulation

Two HCI peers were called upon to assist with the interpretation of the data (in the conceptualisation phase - see section 2.6.2.5) for the purposes of triangulation (see section 2.6.2.2). The involvement of peers in the conceptualisation phase is considered to be one means of addressing concerns about the internal validity of the study. Therefore, the characteristics of these human instruments are also a relevant consideration. These peers, are coded as 'Analyst A' and 'Analyst B'. Analyst A has a PhD in Applied Psychology and over 10 years industrial/commercial experience in HCI design in software development. Analyst B has an MSc in Occupational Psychology and started work as an HCI designer/researcher at SFK a year after this research began.

2.8.5 Validity

This section describes methodological considerations made with respect to construct validity, internal validity, external validity and reliability.

Construct validity for the study has been addressed using a multi-method approach to data collection (see section 2.7.4.1). Firstly, multiple sources of evidence have been utilised including observations made as a participant, ethnographic interviewing (informal) and analysis of project documentation. Secondly, some of the informants from the study (particularly management of SFK) reviewed drafts of the various write-ups.

Internal validity was also addressed by attempting to minimise reactive effects. This was done through several means: firstly, a longitudinal participant-observation method was employed; the skills of the human instrument were well matched to the setting studied; data analysis made use of peer review and interpretation of data (see section 2.7.4.2).

This study was not designed to show results which could be generalised to similar settings, although it is believed that the setting studied is typical of small software development organisations. Thus, external validity (generalisable findings) was not the aim of this study, rather the research question aimed to take an in-depth look at a particular case. Further research would be necessary to formally demonstrate the level to which the findings of this research are generally applicable. One supervisor for this research worked in a similar domain in another organisation and provided some confirmation of external validity during annual review meetings (see section 2.7.4.3). Further informal steps were taken by the researcher to gauge likely general relevance of findings, including meeting with personal contacts in various software organisations.

To address reliability concerns, the documentation of the procedures has been designed so that other researchers could theoretically repeat the studies and arrive at the same results (see section 2.7.4.4). This replication is described as theoretical, because with an in-depth ethnographic study it would not be possible to repeat a study exactly, due to experiential effects on participants in the setting.

2.8.6 Procedure Adopted – The Funnel Approach

Section 2.8.2 begins to describe the early stages of the research. As the research commenced, I was immediately assigned to 'work' as the HCI designer on SFK's main software project (project 1). Data collection therefore started from day one.

The 'funnel' technique (see 2.6.2.3) describes the evolving data collection and analysis stages that were adopted. The following sections describe the layers in the funnel, grouped into sections proposed by Spradley (1980): 'descriptive observations', 'focused observations' and 'selective observations'. Figure 2.1 provides an overview of the funnel focusing technique.

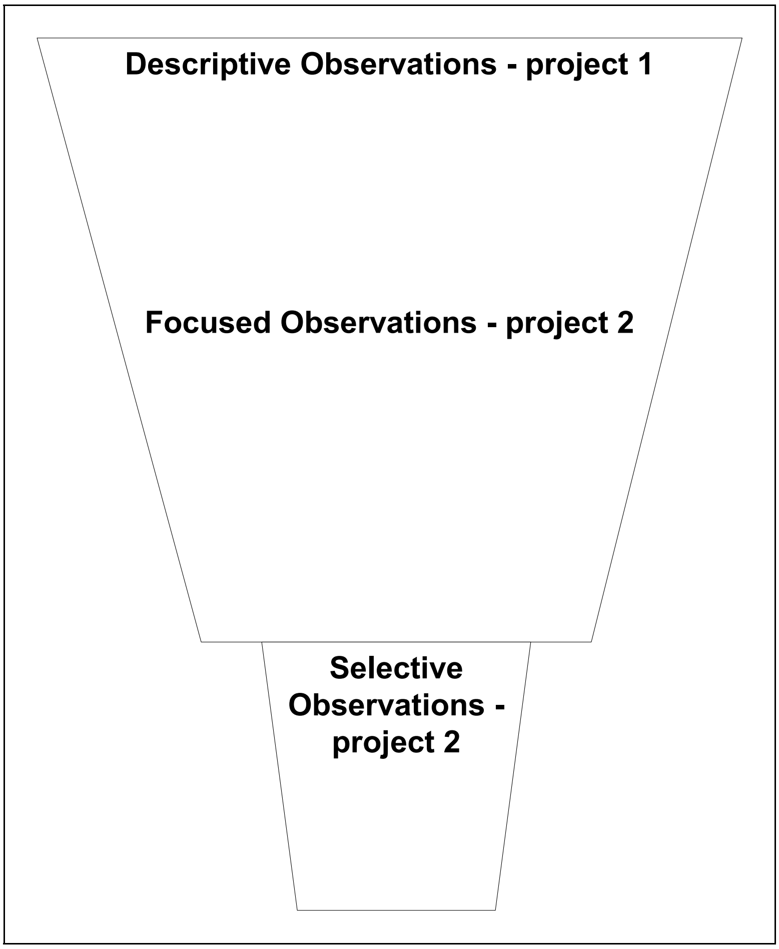
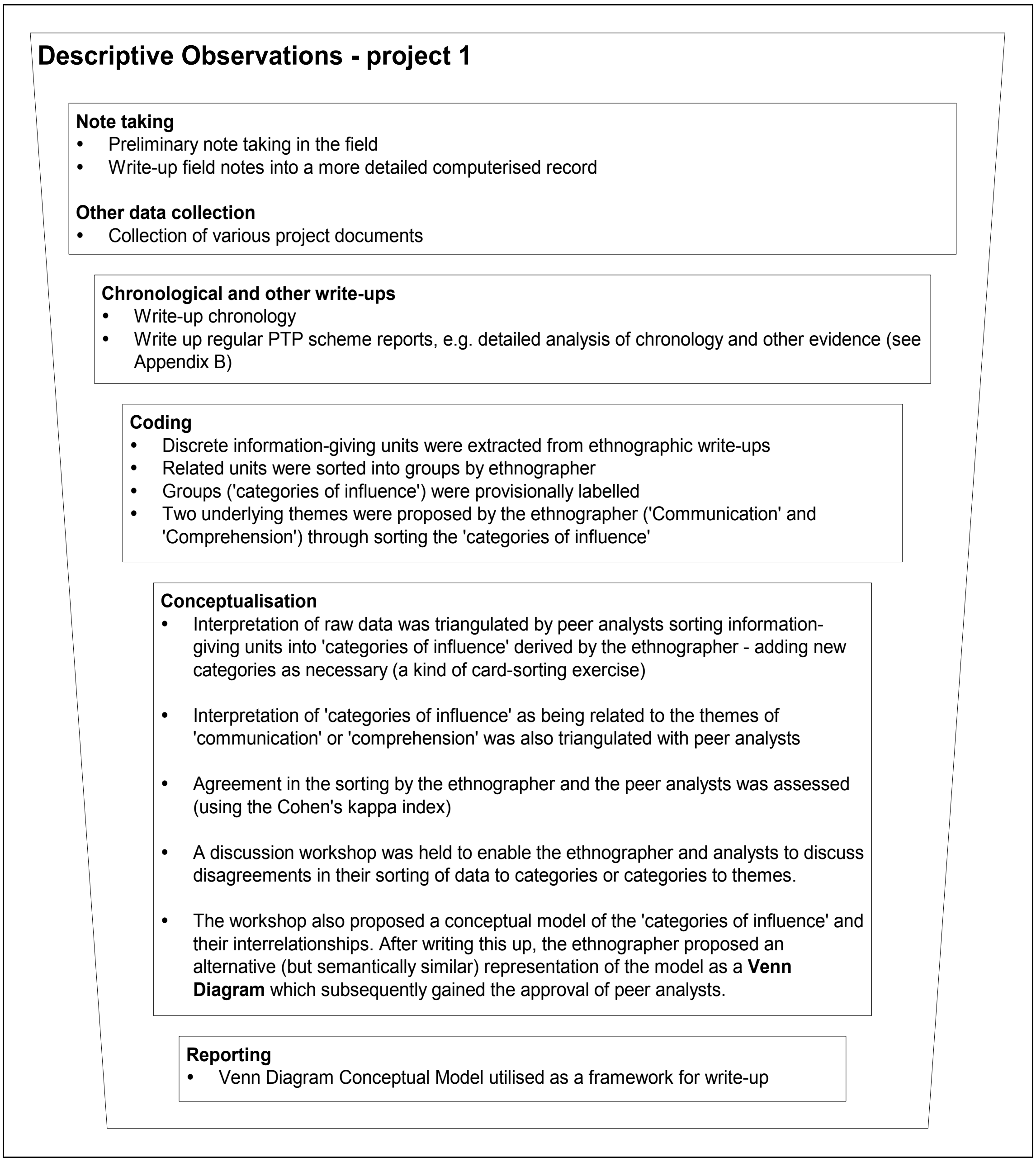

Descriptive Observations - project 1
Focused Observations - project 2
Selective Observations - project 2

Figure 2.1 Overview of funnel technique for focusing ethnographic data collection and analysis (adapted from Spradley, 1980)

2.8.6.1 Descriptive Observations

Descriptive observations for this research were made during the ethnographic study of the ‘project 1’ software development at SFK. The role of HCI designer was performed by the researcher as participant-observer over this year-long project.

Figure 2.2 depicts the gradual focusing of data collection and analysis as progression through the descriptive observations section of the funnel.



Descriptive Observations - project 1

Note taking
- Preliminary note taking in the field
- Write-up field notes into a more detailed computerised record

Other data collection
- Collection of various project documents

Chronological and other write-ups
- Write-up chronology
- Write up regular PTP scheme reports, e.g. detailed analysis of chronology and other evidence (see Appendix B)

Coding
- Discrete information-giving units were extracted from ethnographic write-ups
- Related units were sorted into groups by ethnographer
- Groups ('categories of influence') were provisionally labelled
- Two underlying themes were proposed by the ethnographer ('Communication' and 'Comprehension') through sorting the 'categories of influence'

Conceptualisation
- Interpretation of raw data was triangulated by peer analysts sorting information-giving units into 'categories of influence' derived by the ethnographer - adding new categories as necessary (a kind of card-sorting exercise)
- Interpretation of 'categories of influence' as being related to the themes of 'communication' or 'comprehension' was also triangulated with peer analysts
- Agreement in the sorting by the ethnographer and the peer analysts was assessed (using the Cohen's kappa index)
- A discussion workshop was held to enable the ethnographer and analysts to discuss disagreements in their sorting of data to categories or categories to themes.
- The workshop also proposed a conceptual model of the 'categories of influence' and their interrelationships. After writing this up, the ethnographer proposed an alternative (but semantically similar) representation of the model as a **Venn Diagram** which subsequently gained the approval of peer analysts.

Reporting
- Venn Diagram Conceptual Model utilised as a framework for write-up

Figure 2.2 Breakdown of data collection and analysis for the descriptive observation section of the funnel

2.8.6.1.1 Note taking

As is typical in participant observation, the preliminary data collection took the form of regular note taking. Brief notes were recorded in a project notebook (of the type used by most people in the setting) throughout the observation period and were later written up in a computerised record.

2.8.6.1.2 Other Data Collection

Project documentation was collected throughout the period of study. This included documents that I was provided with and expected to use for guidance for my participatory work as an HCI designer, these included:

Specmaster Demonstrator Task Analysis (Boddy, 1991 December)
Object-Action Analysis (Boddy, 1992, January)
User Interface Rationale (Boddy, 1992, April)
User Interface Design – Specmaster the Discovery (Boddy, 1992a, September)
Specmaster Seed User Interface Plan (Boddy, 1992b, September)

Other essential documents included those which were produced, or contributed to, as part of my role as an HCI designer, such as:

Specmaster Seed Base System Software Specification (Ferrans, Males, Myhill, Skilling, Titcombe, Yates, 1993, April)
The Specmaster Seed System – Task Analysis (Myhill, 1992, September)
The Specmaster Seed System – Non-Functional Requirements (Myhill, 1992, October)
The Specmaster Seed System – Object Model (Myhill, 1992, November)

These documents formed an essential dimension of the ethnographic 'picture' of the activities of the group studied. The use of multiple sources of evidence is advocated as a means to improve the validity of an ethnographic study (see section 2.7.4.1). However, with this research, documentary sources formed an essential aspect of the setting studied and had a clear importance to members of the group studied.

2.8.6.1.3 Chronological and other Write-up

The PTP scheme which funded the research, demanded regular reports throughout the research period (e.g. Myhill, 1993 September; Myhill, 1994 August; Myhill, 1995 May). These reporting requirements provided an opportunity to compile the notes from the ethnography into a more structured format. Another benefit of the PTP scheme requirements was that the reports had to be reviewed by an academic supervisor for Cranfield University, a senior manager from SFK and an expert from a similar domain in another organisation. This provided several contributions to the validity of the research (see sections 2.8.5 and 2.8.2). The academic supervisory review helped to direct the researcher's activities, essential to an ethnographic approach where the researcher has assumed the role of full participant. The review by an SFK manager corresponded to a key informant review as suggested by Yin (see section 2.7.4.1), providing a contribution to construct validity. The review by an expert from a similar domain contributed to an understanding of the external validity of the findings. Thus, the reporting and review requirements of the PTP scheme provided an excellent framework for an ethnographic participant-observation approach.

Extracts from the PTP scheme reports formed the basis of the chronological description of the participant-observation for project 1 (reproduced in Appendix A). A later report provided a more detailed structured analysis of the chronology and other data (see Appendix B).

2.8.6.1.4 Coding the Data

The coding process began using a method drawn from content analysis (see section 2.5.3.1). The raw data comprised write-ups described in section 2.8.6.1.3 and computerised field notes. Using cut-and-paste features of a word processor, discrete information-giving units were snipped from the data in its various forms and were pasted into a single document. The result was a document containing over 200 discrete sentences and paragraphs, for example:

> A Gantt chart style project plan reinforced the waterfall lifecycle approach to the development and served to allocate tasks to individuals. **(Research method – document analysis; Project phase – code design and implementation (early stages); Data Source – field notes September 1992 – May 1993)**

(Appendix C section 1.3.1 contains a reduced sample of these information-giving units, showing how they were grouped by peer interpreters. Only a reduced number appear, because there were too many to be dealt with by peers during the triangulation stage, see section 2.9.1.3)

The next stage of the coding process involved the researcher grouping the discrete information giving units together. The researcher's field experience guided the groupings, so interpretation was an aspect of deciding which units went together. This laborious process produced 21 groupings (see Appendix C section 1.1). Each group was then labelled and a collective term was introduced to describe the set – the groups were referred to as 'categories of influence'. The 'influence' part of the term related to the fact that each of the group labels appeared to be related to some positive or negative influence on the introduction of HCI considerations into a conventional software team.

The process of deriving the 'categories of influence' led the researcher to believe that there were two themes underlying all of the categories. These themes were thought to be 'communication within the team' and 'comprehension (shared mutual understanding, or conceptual integrity) within the team'. To explore this hypothesis, the researcher sorted all of the 'categories of influence' under these theme headings but also allowed for sorting into 'Both' and 'Neither' (see Appendix C section 1.3.3).

Thus, the preliminary coding carried out by the researcher produced a preliminary interpretation of the data as shown in figure 2.3.

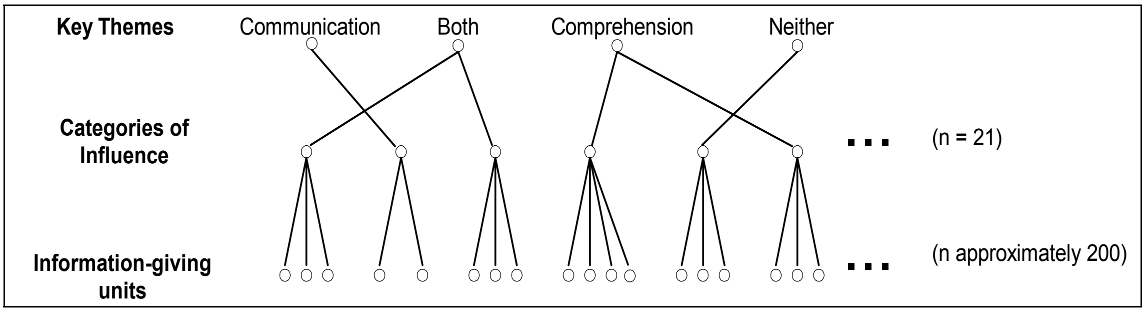


Figure 2.3 Depiction of preliminary coding by the researcher

2.8.6.1.5 Conceptualising the Data

There is clearly considerable crossover in recognised definitions of coding and conceptualisation phases of data analysis (see sections 2.6.2.4 and 2.6.2.5). Deriving preliminary categories and themes through the researcher's preliminary interpretation has been described as coding in the previous section. This section describes further analysis and interpretation of the preliminary categories and themes.

Triangulation was utilised to provide an alternative interpretation of the information-giving units (raw data). Two peer analysts were enlisted to triangulate the data interpretation leading to the preliminary 'categories of influence' and 'themes' proposed by the researcher. Both peers were HCI practitioner/researchers, one with a considerable degree of understanding of the specific context of these studies and the other having some degree of familiarity with the context (see section 2.8.4)

In preparation for the analysis, the 'information-giving units' used by the researcher in the derivation of the categories and themes were printed out on strips of paper, one unit per strip. One third of the units was randomly selected, ensuring that each category was represented by at least two of the units the researcher suggested were related to it. Using all 200 units of evidence would not have been feasible due to time constraints.

Each analyst was separately asked to sort the units into one of the 21 categories created by the researcher. It was also made clear to analysts that they could create new categories, or reject information-giving units if necessary (instructions to triangulating analysts are reproduced in appendix C section 1.2). This card-sorting style approach is very similar to that taken by Turner (1994) – see section 2.6.2.5. It is also very similar to that taken by the researcher in the preliminary interpretation of the data and creation of categories. The only differences were that the triangulating analysts had fewer units to sort and had pre-defined categories in which to sort them. Labels attached to the 'categories of influence' proposed by the researcher were deliberately left coarse, to avoid the possibility of leading the analysts in their interpretation. After information-giving units had been considered, analysts were invited to sort each 'category of influence' into the 'themes' proposed.

The level of agreement between the researcher's interpretation of the data and the analysts' interpretations was assessed by analysing the agreements in the sorting of units into categories, and then categories into themes. Cohen's kappa index was utilised for assessing this inter-rater concordance. From Hays (1988), Cohen's kappa is given by:

$$\mathrm{K} = \frac{N \sum_i x_{ii} - \sum_i x_{i+}\, x_{+i}}{N^2 - \sum_i x_{i+}\, x_{+i}}$$

Where,

x_{ii} represents the number of agreements about category i

x_{i+} represents the number of times judge 1 used category i altogether

x_{+1} represents the number of times judge 2 used category i

N is the number of things rated

Although Cohen's kappa is intended for the analysis of concordance between two independent raters, it was deemed appropriate for this analysis, in order to achieve an indication of the levels of agreement that existed between three raters (the researcher and two analysts). In fact, by analysing a measure of agreement between pairs of judges, some explanation of the differences between levels of agreement could be offered.

Interpretation of Cohen's kappa indices of agreement can be assisted by two important considerations.

1. If agreement between two raters is exactly what would be expected under independence, then k will be zero. If there is perfect agreement between raters, k will be exactly one (Hays, 1988).

2. A kappa value greater than 0.60 is regarded as demonstrating substantial agreement, values below 0.20 are considered poor and values between 0.21 and 0.60 are considered fair to moderate (Landis and Koch, 1977).

The coarse labelling of the categories and the restrictive nature of the card-sorting exercise provided a very limited means for the triangulating analysts to contribute to the conceptualisation of the data. Therefore, a discussion workshop was held with the researcher and both analysts. The purpose of the workshop was to discuss any disagreements in the card-sorting in order to ascertain whether the levels of agreement suggested by Cohen's kappa were accurate. The workshop discussion graduated towards the production of a conceptual model (or, an overall picture) of the 'categories of influence' and how they were believed to interrelate. After writing-up the conceptual model produced at the workshop, I began to refine the model. After a few attempts, the idea of using a Venn Diagram representation struck me. This seemed to provide a very neat representation of the model produced during the workshop and was semantically very similar. Checking back with the analysts confirmed my belief that this was indeed a very good representation of the model we produced during the workshop and the surrounding discussion.

Using a Venn Diagram as a representation of ethnographic data perhaps has no precedent. However, 'charting' or producing an abstract representation of the data as a whole is a usual aim of the conceptualisation phase of ethnographic data analysis (see section 2.6.2.5).

2.8.6.1.6 Reporting

The Venn Diagram conceptual model of the data was utilised as a framework for reporting the findings of the ethnographic study of project 1, forming the conclusion of the 'descriptive findings' stage of the funnel approach.

2.8.6.2 Focused Observations

At the end of 'project 1', an opportunity arose to extend the ethnographic research to cover a further year-long software project with largely the same development team. This opportunity offered a number of important advantages:

- It opened the door for the researcher to focus the scope of the investigation according to the 'categories of influence' and Venn Diagram conceptual model from the 'descriptive observations' stage. The opportunity therefore allowed progression down the 'funnel' to the 'focused observations' and 'selected observations' stage.

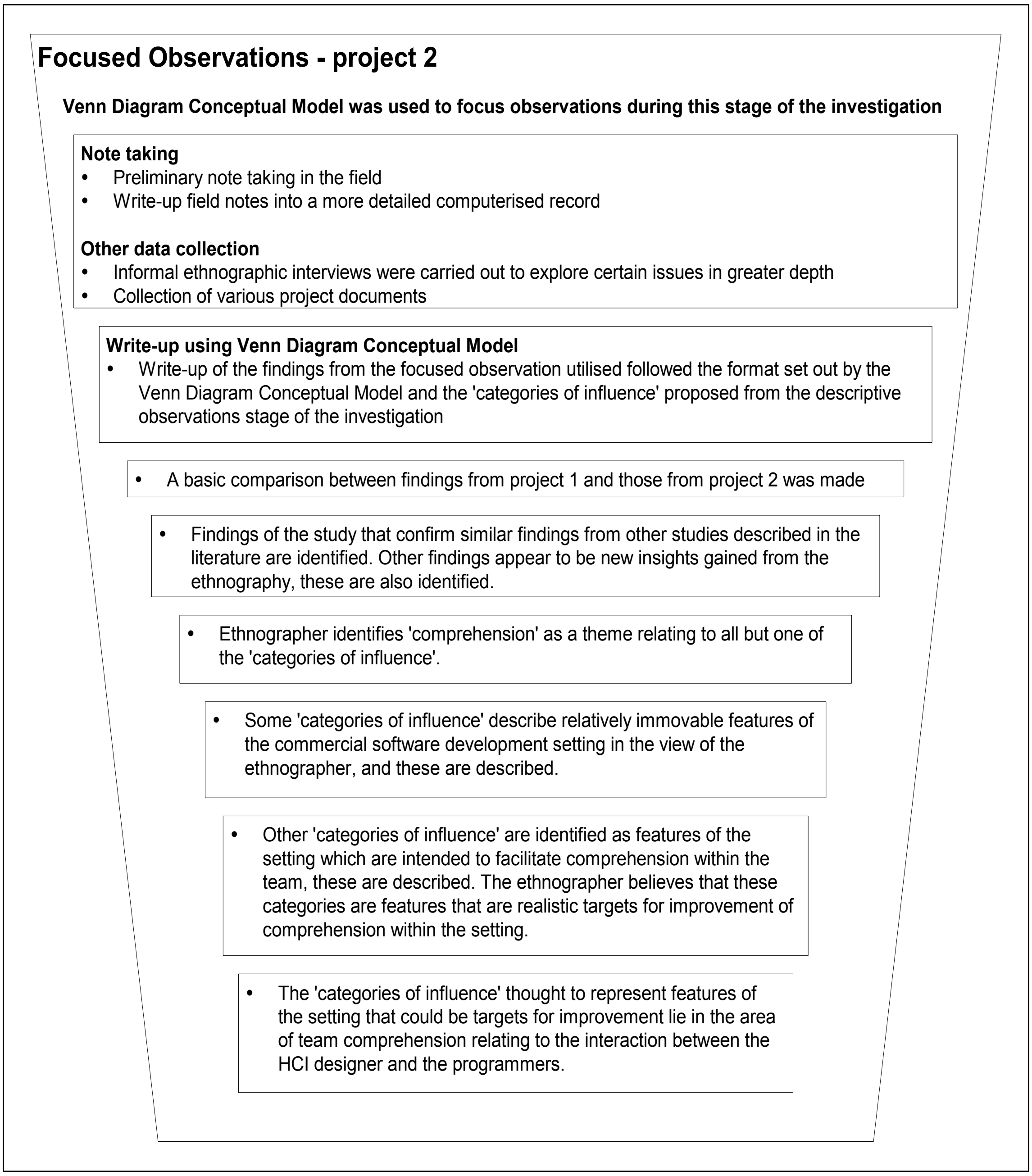
- Findings from project 1 could be confirmed through the investigation of another commercial software project with slightly different features – thereby providing a wider exploration of the issues, contributing to external validity of the results.

- Investigation of a second major project for SFK removed several potential biases which may have affected the investigation of project 1, including:
  - the researcher's inexperience in the HCI designer role;
  - team members' unfamiliarity with the HCI designer role;
  - the developers' unfamiliarity with Microsoft Windows, C++ and Object-Oriented programming;
  - team members' unfamiliarity with each other.

Perhaps the most important of the biases addressed by the second study is the removal of 'knee-jerk' reactions of team members' to the introduction of HCI design. Further investigation without this extraneous effect in particular was regarded as worthwhile.

Therefore, the opportunity to extend the participant-observation to cover another project was taken. Initially, the investigation was focused on aspects of the project, the people, and the setting relating to the 'categories of influence' identified from the project 1 investigation.

Figure 2.4 depicts the continuing focusing of data collection and analysis as progression through the focused observations section of the funnel.



Focused Observations - project 2

Venn Diagram Conceptual Model was used to focus observations during this stage of the investigation

Note taking
- Preliminary note taking in the field
- Write-up field notes into a more detailed computerised record

Other data collection
- Informal ethnographic interviews were carried out to explore certain issues in greater depth
- Collection of various project documents

Write-up using Venn Diagram Conceptual Model
- Write-up of the findings from the focused observation utilised followed the format set out by the Venn Diagram Conceptual Model and the 'categories of influence' proposed from the descriptive observations stage of the investigation

- A basic comparison between findings from project 1 and those from project 2 was made

- Findings of the study that confirm similar findings from other studies described in the literature are identified. Other findings appear to be new insights gained from the ethnography, these are also identified.

- Ethnographer identifies 'comprehension' as a theme relating to all but one of the 'categories of influence'.

- Some 'categories of influence' describe relatively immovable features of the commercial software development setting in the view of the ethnographer, and these are described.

- Other 'categories of influence' are identified as features of the setting which are intended to facilitate comprehension within the team, these are described. The ethnographer believes that these categories are features that are realistic targets for improvement of comprehension within the setting.

- The 'categories of influence' thought to represent features of the setting that could be targets for improvement lie in the area of team comprehension relating to the interaction between the HCI designer and the programmers.

Figure 2.4 Breakdown of data collection and analysis for the focused observation section of the funnel

Primary data collection proceeded as before with daily note-taking during the observation period and a write-up in a computerised record later. Project documentation also formed part of the data collected. During the investigation, informal ethnographic interviews were conducted in a way that would remind the team members of my research role.

The Venn Diagram conceptual model of the data, and the ‘categories of influence’ it represents, formed the framework for the write-up of the ethnography.

An analysis of the findings so far was then carried out. Firstly, a basic comparison of project 1 and project 2 was made.

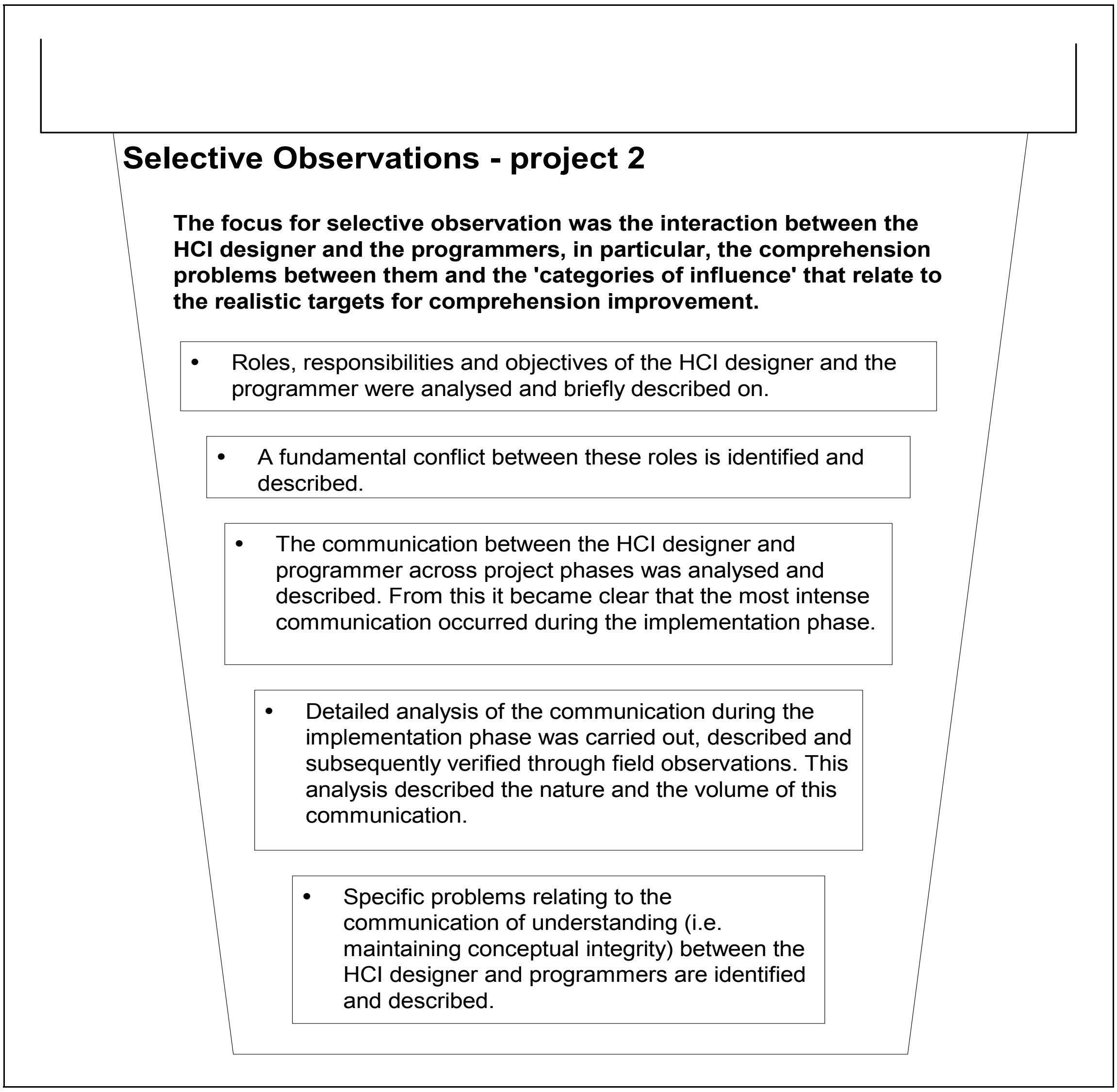
Then, the researcher identified a primary underlying connection of all of the 'categories of influence' – that most related to some aspect of comprehension within the team. The team's understanding of HCI design intent was one aspect of this comprehension. The researcher suggested that some categories described features of the commercial software development setting that had an effect on comprehension within the team. Finally, categories that described features of the setting which intended to facilitate understanding within the team were identified. From the researcher's experience, it was these features that could provide realistic targets for the improvement of comprehension within the setting. Both the apparently immutable features of commercial software product and the 'categories of influence' representing targets for improvement, were related back to existing literature, where it existed, or were highlighted as new findings from this investigation.

It is apparent that the primary stakeholders for the concepts expressed as realistic categories for improvement of comprehension are the HCI designer and the programmer in the software team. Thus, the very bottom of the 'focused observations' section of the funnel draws focus onto the interaction between the HCI designer and the programmers, in particular, the comprehension problems between them and the 'categories of influence' that relate to the realistic targets for improvement.

2.8.6.3 Selective Observations

Selective observations were made during the study of project 2 and followed a natural focus from the analysis and write-up activities described in the 'focused observations' section above. The focus for the selective observations is therefore the interaction between the HCI designer and the programmers, especially the comprehension problems between them.

Figure 2.5 depicts the final focusing of data collection and analysis as progression through the selective observations at the end of the funnel.



Selective Observations - project 2

The focus for selective observation was the interaction between the HCI designer and the programmers, in particular, the comprehension problems between them and the 'categories of influence' that relate to the realistic targets for comprehension improvement.

- Roles, responsibilities and objectives of the HCI designer and the programmer were analysed and briefly described on.

- A fundamental conflict between these roles is identified and described.

- The communication between the HCI designer and programmer across project phases was analysed and described. From this it became clear that the most intense communication occurred during the implementation phase.

- Detailed analysis of the communication during the implementation phase was carried out, described and subsequently verified through field observations. This analysis described the nature and the volume of this communication.

- Specific problems relating to the communication of understanding (i.e. maintaining conceptual integrity) between the HCI designer and programmers are identified and described.

Figure 2.5 Breakdown of data collection and analysis for the selected observations at the end of the funnel

Skills and techniques learned as participant HCI designer within in a software team provided the researcher with an appropriate means of focusing on the interaction situation between the HCI designer and programmers.

Firstly, the roles, responsibilities, and objectives of the HCI designer and the programmer were analysed and briefly outlined. These descriptions were made on the basis of the researcher's long field-experience of these roles and were written alongside ongoing participant-observation, allowing some verification of the interpretation.

The outlined roles and responsibilities as well as field experience led to the strong belief that there is a fundamental conflict in the priorities of these roles. This conflict is then described.

Although comprehension was the focus of these selected observations, it was the communication between the HCI designer and the programmers across project phases that was analysed first, as this was the vehicle for their mutual comprehension.

It was observed that the communication between these roles was at its most intense during the implementation phase, so the analysis focused on this phase. Further field observation facilitated description and categorisation of the nature and volume of this communication.

At the very base of the funnel, based on ethnographic observation and long field experience, the specific problems relating to the interaction between the HCI designer and the programmers are identified and described.

The final discussion introduces potential solutions to the specific problems found at the base of the funnel.

2.9 Results and Discussion

This section presents the results and discussion of the ethnographic investigation of two year-long software projects, with the researcher participating in the software team as an HCI designer.

2.9.1 Descriptive Observations of the Project 1 Setting Following the Introduction of an HCI Designer into the Software Team

This section covers the 'descriptive observations' of the ethnographic study (see section 2.8.6.1 for a further description of this aspect of the methodology) and describes the findings from the investigation into the effects of introducing an HCI designer into a project team. A description of how the research began and the PTP scheme is provided in section 2.8.2. This section begins by briefly describing the host company for the ethnography, before introducing software project 1. Data analysis proceeded according to the methodology outlined in section 2.8.6.1. Findings from the conceptualisation stage of the analysis (including triangulation with peers) are reported in detail in section 2.9.1.3. Finally, a conceptual model is introduced to describe the main features of the setting studied. This model is then used as a framework to describe the observations made by the researcher (section 2.9.1.6).

2.9.1.1 Introduction to SFK Technology Limited

The core business of SFK Technology Limited (the software subsidiary of British Hydromechanics Research Group (BHRG)) is selling software products and services to process engineering and water industries, as well as engaging in software and engineering related research projects, particularly for the European Commission.

SFK was formed in 1991 from the amalgamation of BHRG's software division and the software development team of a company called Amtech. This research started in September 1992, not long after the company was formed.

The management of SFK gave high regard to organisational and cultural issues, considering this an important element of effective software development. The result of this was that SFK had an ostensibly 'Object Oriented' structure[1] and a culture that emphasises the development, and contribution, of the individual within a team context. The company is made up of around 24 staff, all but one were graduates and a large proportion also held postgraduate qualifications. Approximately half of the technical staff were aged 30 or less but several people were older and had worked in computing since the 1970s.

[1] This is analogous to a matrix organisational structure

SFK became interested in supporting an ethnographic study involving the introduction of an HCI designer role into their software team, because they were keen to produce software with a greater user-centred emphasis.

2.9.1.2 Introduction to Software Project 1

Software project 1 was SFK's first major software project to include an HCI designer (the participant ethnographer) as a full member of the development team. Thus, it was the first time that responsibility for the user interface was explicitly given to an HCI designer. Previously, responsibility for the user interface was tacitly distributed amongst the programmers.

The project 1 software development aimed to utilise knowledge-based systems (KBS) technology to assist mechanical engineers with the selection and specification of process plant equipment.

Some characteristics of project 1 serve to illustrate the kind of software venture embarked upon (see Table B.1 in appendix B for further details):

- there was no client for the software, the venture was speculative;
- the project was also complex and innovative; no similar software was believed to exist in the marketplace;
- SFK's software personnel had a limited understanding of the user domain, which was the selection and specification of control valves for new process plant designs, by mechanical engineers. In recognition of this limitation, knowledge engineers were also brought onto SFK's development team for the first time;
- the team included an HCI designer, another first for an SFK development team;
- the technical core of the team (the programmers and IT designers) had very little experience of the software technology to be used on the project, which included, Microsoft Windows programming, C++ and Object-Oriented design;
- the project team consisted of between eight and ten people;
- the project budget was approximately £250,000;
- project duration was one year.

These project characteristics set the scene for the software development and the ethnographic participant-observation findings reported in the remainder of section 2.9.1.

2.9.1.3 Conceptualisation of the data

The conceptualisation stage of the data analysis developed a conceptual framework to aid the interpretation of the ethnographic data. This section describes and discusses the results of the conceptualisation stage (refer to section 2.8.6.1.5 for a description of the methodology followed during this stage, and the earlier sections of 2.8.6.1 for more detail of the analysis preceding this stage).

Initially, the data were coded as described in section 2.8.6.1.4. Essentially, the researcher created 21 'categories of influence' by sorting 200 information-giving units into related groups, which were then labelled (see appendix B section 1.1). The researcher then suggested that each of the 21 categories related to underlying themes of 'communication' or 'comprehension' – the 21 categories were then sorted accordingly (see appendix C section 1.3.3). In order to triangulate this interpretation of the data, two peer analysts were called upon. A reduced selection of information-giving units (randomly selected) was presented to each peer analyst in turn. Analysts were asked to sort the units into the categories that had been proposed and to create new categories or reject units as they saw fit. Next, the analysts were asked to sort the categories under headings 'communication', 'comprehension', 'both' or 'neither'.

After the sorting, it was necessary to examine the sorting agreements between the researcher and the two peer analysts in order to determine the extent to which the three interpretations differed. The Cohen's kappa index was utilised for assessing this level of agreement and thereby facilitated the triangulation of the data interpretation. Table 2.2 summarises Cohen's kappa indices calculated for this triangulation.

Table 2.2 Summary of Cohen's kappa measures calculated for the triangulation of data interpretation

Analyst Pair **Rating**	Ethnographer – Analyst A	Ethnographer – Analyst B	Analyst A – Analyst B
Agreement sorting 'units of evidence' into 'categories of influence'	0.49	0.69	0.51
Agreement sorting 'categories of influence' into 'key themes'	0.13	0.38	0.23
Agreement sorting 'categories of influence' into 'key themes' but accounting for non-exclusivity of 'key theme' category	0.26	0.51	0.44

High levels of agreement shown by kappa indices between 0.49 and 0.69 occurred when analysts sorted the 'units of evidence' into 'categories of influence' (see table 2.2). The 'Ethnographer - Analyst B' kappa index of 0.69 was perhaps understandably the highest as both worked in similar software teams and had an understanding of the specific problem context. However, the 'Ethnographer - Analyst A' kappa index also showed a relatively high level of agreement, even though Analyst A did not have an equivalent deep knowledge of the specific context.

When analysts sorted the ‘categories of influence’ into ‘key themes’ of ‘Communication’ and ‘Comprehension’, Cohen’s kappa indices between 0.13 and 0.38 were evident. These results pointed towards the rejection of the hypothesis that ‘Communication’ and ‘Comprehension’ were valid ‘key themes’ underlying the ‘categories of influence’. However, in this analysis partial agreement between judges was not accounted for in the Cohen’s kappa indices. Partial agreement occurred because the ‘key themes’ classifications were non-exclusive, i.e. the classifications were ‘Communication’, ‘Comprehension’, ‘Both’ or ‘Neither’. Therefore, if one analyst sorted a ‘category of influence’ into the classification ‘Both’ and another analyst selected the category ‘Communication’, this could be considered as half an agreement. Cohen’s kappa indices were recalculated assigning half marks to represent partial agreements. The resulting Cohen’s kappa indices ranged from 0.26 to 0.51, which was considered to be only a slight improvement.

The results of the Cohen’s kappa analysis of agreement were followed up during the discussion workshop held by the researcher with the two peer analysts. During the discussion, it transpired that there was a greater degree of agreement between analysts (including the researcher) than the Cohen’s kappa indices suggested. Descriptions of the ‘categories of influence’ were deliberately coarse in order to avoid the researcher introducing bias by leading the peer analysts. Thus, some results of the triangulation were clearly influenced by the interpretation each of the analysts had made of the ‘category of influence’ description. This became evident when some pairs of categories were obviously strongly linked. For example, any disagreement in units sorted into category 18 (effects of team members’ personalities) could be explained by considering units sorted into category 3 (the technical core of the team were motivated to resist change). Thus, if one analyst sorted a unit into category 18 and another analyst sorted the unit into a different category, this different category was always category 3. So, if category 3 and 18 were combined, all analysts would agree on the units to sort into the new category. Further analysis to determine the precise effect on the kappa indices was deemed unnecessary because the workshop session clearly showed that by removing ambiguity, higher indices of agreement would be found.

The discussion workshop also covered the sorting of ‘categories of influence’ into ‘key themes’. From the discussion it was clear that analysts could not agree on the assignment of the ‘categories of influence’ to the ‘key themes’ proposed. Workshop discussion and supporting Cohen’s kappa indices ultimately led to the rejection of the ‘key themes’ notion.

The conclusion of the workshop supported the ‘categories of influence’ hypothesised but rejected the hypothesis of two underlying themes.

A second aspect of the workshop discussion focused on the ‘categories of influence’ and their interrelationships. A loosely hierarchical conceptual model was constructed during the workshop showing ‘categories of influence’ and their key interrelationships. An accurate depiction of this is shown in appendix C figure C.4.

During the later write-up, the researcher decided to represent the conceptual model as a Venn Diagram. Subsequent review with the peer analysts confirmed the semantic similarity to the conceptual model produced during the workshop.

2.9.1.4 Overview of the Venn Diagram Conceptual Model of Categories of Influence

A Venn Diagram[2] conceptual model was developed as an abstract representation of the data associated with the descriptive observations stage of the ethnographic investigation. This conceptual model forms the framework for reporting the findings of the ethnographic study of project 1. This section provides an overview of the Venn Diagram model.

Although a great many of the 'categories of influence' modelled could be viewed as interrelated, the Venn Diagram representation (figure 2.3) provided a useful framework for outlining the **dominant** relationships and **overall** classification of 'categories of influence' considered. In other words, the diagram is not strictly a Venn Diagram as many of the categories could not be considered mutually exclusive. The diagram provides a symbolic conceptual model of 'categories of influence' which arose following a software company's attempt to increase the user-centred emphasis of their software through the introduction of an HCI designer into the software team. The model is derived from the ethnographic study of the project 1 software development.

The 'universe of discourse' for the Venn Diagram is the domain of complex software projects produced in a commercial development environment. The complexity of a software project is fundamental to this model, because if the software were simple, the majority of the 'categories of influence' may not become apparent. Demands on software production created by commercial influences also provide strong boundaries to the universe of discourse.

There are three main sets shown in this Venn Diagram: People Issues (P), Representation Issues (R) and Lifecycle Issues (L). The definition of these sets came from branches on the original hierarchical conceptual model produced during the researcher/analysts discussion workshop.

The 'People issues' set includes 'categories of influence' relating to individuals (team members), the software team and management influence.

[2] Mathematical set notation adopted is taken from Ledermann, Hilton, Jackson, Jenkins, MacHale, Stewart, Tall, Trustrum, Unsworth, Vajda, Williams, and Wylie (1980)

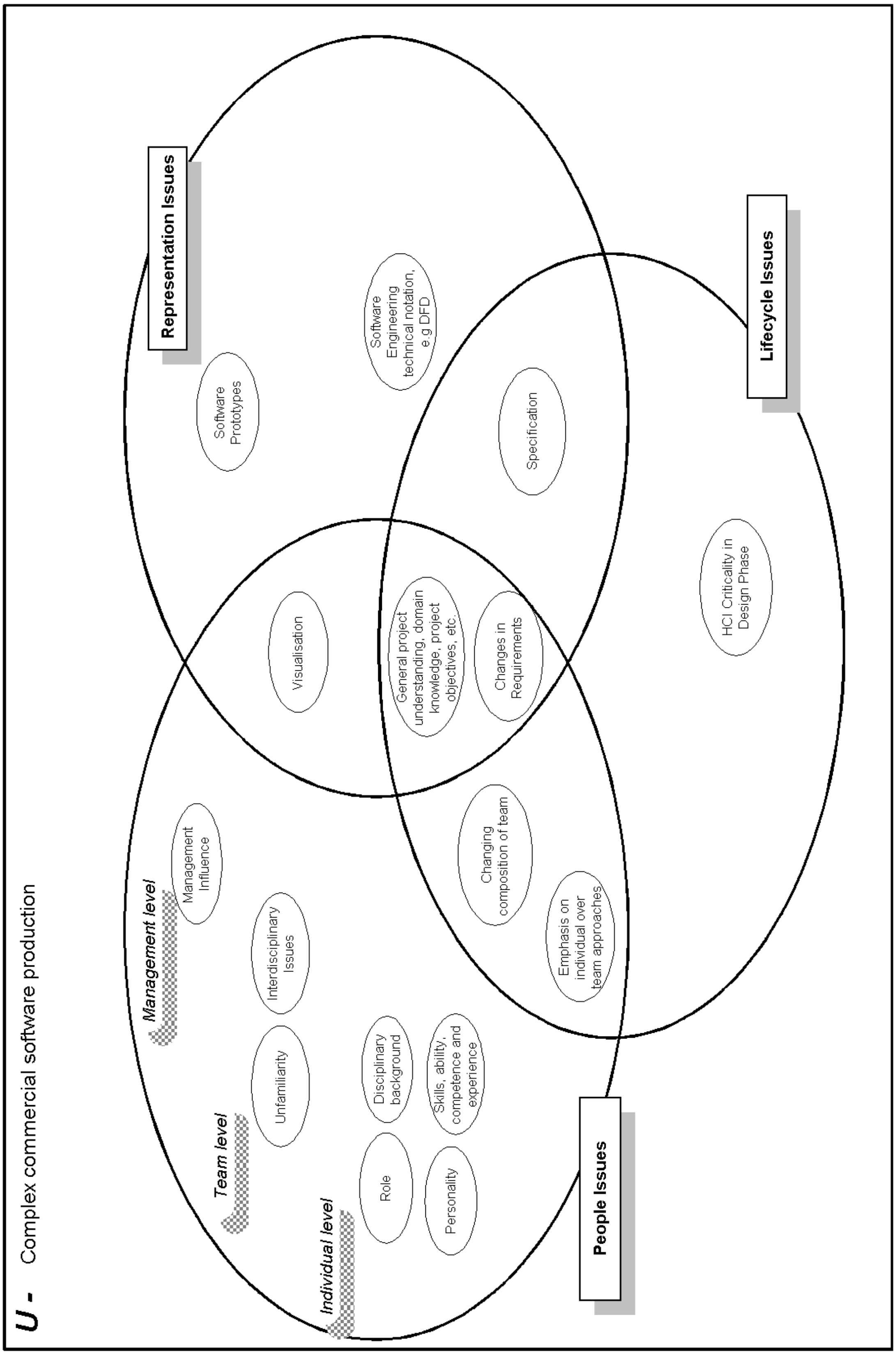

U - Complex commercial software production
Representation Issues
Lifecycle Issues
People Issues
Software Prototypes
Software Engineering technical notation, e.g DFD
Specification
HCI Criticality in Design Phase
Visualisation
General project understanding, domain knowledge, project objectives, etc.
Changes in Requirements
Changing composition of team
Emphasis on individual over team approaches
Management level
Management Influence
Interdisciplinary Issues
Team level
Unfamiliarity
Disciplinary background
Skills, ability, competence and experience
Individual level
Role
Personality

Figure 2.6 Venn Diagram Conceptual Model of ‘Categories of Influence’ arising through the introduction of an HCI designer role into a software team

The ‘Representation issues’ set includes ‘categories of influence’ relating to representation, the understanding of representations and other influences on the process of developing representations.

The ‘Lifecycle issues’ set includes ‘categories of influence’ closely related to the practicalities of commercial software development; these include the need to manage scarce resources and provide milestones to the development process (e.g. the Specification).

2.9.1.5 Data Supporting the Descriptive Observations

The following ethnographic write-up contains ‘related data samples’ as an illustration of assertions and explanations made in the text. Confidentiality concerns restricted the nature and number of the data samples that could be reported. Although individuals within the team have been coded, because the host company was small, direct coding of individuals to a reference identification was an insufficient means of maintaining confidentiality. These ethical concerns have therefore had an impact on the way that data samples can be reported.

Project 1 data samples have been re-coded to report the research method of the inferences; deliberately vague descriptions of the specific people involved; and approximate identification of project phases from which observations and interpretations were drawn. Finally, the two main reports comprising written-up field notes are identified. The following table illustrates the re-coding scheme used in section 2.9.1.6.

Table 2.3 Re-coding scheme adopted for reporting data samples for project 1

Research method	participant-observation ethnographic interview document analysis interpretation
People	senior management sales personnel project manager HCI designer (the researcher) software designer / programmer (senior programmer) knowledge engineer programmer
Project phases[3]	requirements code design HCI design implementation (early stages)[4] implementation (late stages)
Source of Data	field notes September 1992 – May 1993 field notes June 1993 – September 1993

[3] It should be noted that project phases are not strictly linear, for example, the code design and HCI design proceed in parallel

[4] This also encompasses the ‘mid’ stage of implementation. During early and mid stages of development, the pace of implementation is normal, and various team members are beginning the translation from specification and code design, to implementation.

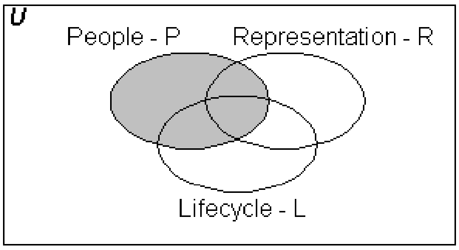
2.9.1.6 Descriptive Observations – Detailed Report of Venn Diagram (project 1)

The Venn Diagram, its three main sets and the intersections between sets have specific meaning in the context of conceptual model. Explanations of the 'categories of influence' are presented as a means to understand the conceptual model and the main findings of the ethnographic study of project 1.

The explanations that follow are organised by the three primary sets, 'People issues', 'Representation issues' and 'Lifecycle issues'. 'Categories of influence' are described under the set they belong to. Some categories appear in the intersection of two or three sets, so these intersections are also used as headings in the explanation that follows.

2.9.1.6.1 People Issues

Set, **P**



'Categories of Influences' which can be considered People Issues

Within the People Issues set, there are three levels of sub-groupings (but not sub-sets necessarily - this is not a formal Venn Diagram). The sub-levels correspond to individual, team and management level 'categories of influence'. Categories, which are in the intersections between sets, may be related to more than one of these levels (these are introduced under headings relating to the intersections).

<u>Individual-level 'categories of influence'</u>

Personality

Aspects of each team member's personality appeared to dictate their effectiveness in a software team or in a particular role. For example, some technical team members appear to take a solitary attitude to work and are not team oriented. Thus, personality can apparently effect the communication effectiveness in a software team and the sharing of team goals, vision and conceptual integrity.

Related data samples

I continue to find it unbelievable that some programmers make assumptions about what their colleagues are doing, rather than talking things through with them. It has transpired that two programmers sitting next to each other, who were working on a part of the system that would fit together, each made assumptions about the approach the other was taking rather than discuss it. When asked, neither of them claimed to know anything about the approach that the other had taken. **(Research method - participant-observation / ethnographic interview; People - senior designer/programmer and a programmer; Project phase – implementation (early stages); Data Source - field notes September 1992 – May 1993)**

A significant proportion of the technical core of the team were observed to have poor social and communication skills and often preferred to work alone. Examples of this included, non-participation and apparent lack of interest in project meetings, unwillingness to seek clarification on various technical points, preferring to make assumptions and general willingness to act unquestionably on the instructions of others. **(Research method – participant-observation / interpretation; People – two senior designer/programmers and three programmers; Project phase – code design, implementation (early & late stages); Data Source - field notes June 1993 – September 1993)**

Younger and more inexperienced team members for example, the HCI designer and the knowledge engineer, out of necessity, took on more responsibility in managing the development, based largely on their social skills. **(Research method – participant-observation / interpretation; People – knowledge engineer & HCI designer; Project phase – code design, implementation (early & late stages); Data Source - field notes June 1993 – September 1993)**

These technical professionals ...(*those in the technical core of the team*)...looked upon each bug as a potential personal affront reflecting on their abilities. **(Research method – participant-observation / Interpretation; People – three senior designer/programmers and three programmers; Project phase – implementation (early & late stages); Data Source – field notes June 1993 – September 1993)**

When bugs were discovered in the implementation, I was surprised at how the programmers took them personally. On discovery of a bug, some programmers were keen to attribute blame for it, seemingly to clear their own name. In some ways this seemed to emphasise the individualist rather than team approach to the development. **(Research method – participant-observation / interpretation; People – three senior designer/programmers and three programmers; Project phase – implementation (early & late stages); Data Source - field notes June 1993 – September 1993)**

Role

Each role in a software team has different responsibilities and priorities. Some of these responsibilities changed with the introduction of the HCI designer role.

Related data samples

Software architects and programmers rarely sought to gain any domain understanding as they saw this beyond the scope of their role **(Research method – participant-observation / ethnographic interview; People – two senior designer/programmers and two programmers; Project phase – code design & implementation (early & late stages); Data Source – field notes June 1993 – September 1993)**

One of the programmers resented the loss of a creative outlet when he could no longer design the UI himself. **(Research method – participant-observation / ethnographic interview; People – programmer; Project phase – implementation (early stages); Data Source - field notes September 1992 – May 1993)**

There were numerous occasions when software architects and programmers were unhappy about an HCI designer showing concern about software structure. **(Research method – participant-observation; People – two senior designer/programmers and two programmers; Project phase – code design; Data Source - field notes September 1992 – May 1993)**

During the software production process, conflicts arising from differing role priorities have an effect on the performance of the team and the software production process. Creative tension between diverse roles is what ultimately generates the software product. For example, programmers must ensure that the code is well structured and bug free and the HCI designer must ensure that the users' needs are satisfied.

Related Data Sample

The responsibilities associated with the roles of the HCI designer and those of the technical core of the team were often in conflict. HCI design activities would regularly complicate the tasks of the technical core of the team for reasons associated with the UI and the user's perspective. Corners that were cut by the technical core of the team were often unacceptable from a user perspective. **(Research method – participant-observation; People – three senior designer/programmers, three programmers and HCI designer; Project phase – code design, HCI design and implementation (early and late stages); Data Source - field notes June 1993 – September 1993)**

Constraints on how a particular role is performed were not uncommon. For example, HCI design effort deemed necessary by the HCI designer may be overruled by the project manager for commercial reasons.

Related data samples

The project manager, motivated to produce the [project 1] system to Time, Budget and Quality (TBQ) was therefore keen to ensure that each designed mechanism would be viable to construct in the time available and be of sufficient quality. **(Research method – participant-observation; People – project manager and HCI designer; Project phase – HCI design and implementation (Early and Late stages); Data Source - field notes June 1993 – September 1993)**

…the specification acted as a contract between the SFK management and the project manager. It was the project manager's motivations and direct influence in particular that contradicted the openly flexible approach to specification. **(Research method – participant-observation/interpretation; People – senior manager and project manager; Project phase – requirements, code design and implementation (early stages); Data Source – field notes September 1992 - May 1993)**

Disciplinary Background

Team members' fundamental perceptions of the overall priorities appeared to be linked with their primary disciplinary background and training. People trained in software engineering and programming generally strive for elegant code (or at least recognise the worth of such) over all other considerations. People trained in HCI or some form of social science are generally less interested in the underlying aspects of the software but are more interested in its interface to the user.

Related data sample

...we (*the programmers and the HCI Designer*) learned to compromise but each party usually felt wronged by compromising their ideals, for me this was the ideal interaction, for programmers the ideal was elegant code. **(Research method – participant-observation; People – three senior designer/programmers, three programmers and HCI designer; Project phase – code design, HCI design and implementation (early and late stages); Data Source - field notes June 1993 – September 1993)**

Skills, ability, competence and experience

Skills, ability, competence and experience dictate the potential performance of an individual in a software team. In software teams, the performance of the individual can have a strong effect on the performance of the team and the overall software production. Effects include differences in programmer performance, team members making poor assumptions and difficulties in comprehending HCI design intent.

Related data samples

...the severity and number of bugs could have been reduced had there been fewer misunderstanding and misinterpretations, greater team cohesiveness and more relevant skills and experience than was evident in the [project 1] team. **(Research method – interpretation; People – three senior designer/programmers, three programmers and HCI designer; Project phase – code design and implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

The majority of team members on the project were lacking in skills or experience in areas fundamental to their roles. **(Research method – participant-observation; People – two senior designer/programmers and two programmers; Project phase – code design and implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

Team-level ‘categories of influence’

Unfamiliarity

Team members who have not worked together before and are unfamiliar with each other appear to experience problems communicating and understanding each other.

Related data samples

Only the technical core of the [project 1] team had worked together before and new roles, skills and diverse disciplinary backgrounds of new team members introduced a certain degree of unfamiliarity. **(Research method – participant-observation; People – project manager, three senior designer/programmers, three programmers, HCI designer, knowledge engineer; Project phase – requirements, code design and implementation (early stages); Data Source – field notes September 1992 - May 1993)**

The lack of familiarity brought with it communication difficulties within the now multi-disciplinary team. **(Research method – interpretation; People – project manager, three senior designer/programmers, three programmers, HCI designer, knowledge engineer; Project phase – requirements, code design and implementation (early stages); Data Source – field notes September 1992 - May 1993)**

Some of the previously senior members of the technical core of the team seemed to be fearful of the younger and more dynamic team members and perhaps even saw them as a threat to their own job security. **(Research method – participant-observation; People – two senior designer/programmers and two programmers; Project phase – code design and implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

Interdisciplinary Issues

Problems appear to arise at a team level when individuals from diverse disciplines are working towards a common goal. With respect to **comprehension,** a specific issue is that *people from diverse disciplines and roles seem to interpret what they see differently*. This means that team members may interpret project knowledge in different ways or they may draw different conclusions from various representations designed to provide a visualisation of the ultimate product. This is a team level ‘category of influence’ as it is very important to have a clearly focused team understanding, or consistent group vision of the software being produced in order to maintain conceptual integrity.

Related data samples

During a presentation of the visual prototype I could not dissuade a particular programmer from focusing their attention on the sample data I had used in the mock-up, rather than the complex interaction demanded by the design. It was as if, their deeply ingrained programming instincts forced them to view the data as the most important part of the job and the interaction as nothing more than secondary. **(Research method – participant-observation/interpretation; People – senior designer/programmer and HCI designer; Project phase – HCI design; Data Source – field notes June 1993 – September 1993)**

Technical team members appeared to derive information about what data was displayed whilst failing to recognise what may be complex or troublesome user interface mechanisms. **(Research method – interpretation; People – two senior designer/programmers, two programmers and HCI designer; Project phase – HCI design and implementation (early stages); Data Source – field notes June 1993 – September 1993)**

Management-level 'categories of influence'

Management Influence

Management exerts influence on the team and other aspects of the software production in order to meet the constraints that technical and commercial pressures provide.

Related data samples

...it is usual good software engineering practice to design code elements to be re-usable but on the [project 1] development some programmers were explicitly told by management not to bother with that. **(Research method – participant-observation; People – project manager, senior designer/programmer and two programmers; Project phase – code design; Data Source - field notes September 1992 - May 1993)**

…the profile of [project 1] and high expectations has probably increased external pressure on the team but this seems to have been offset by a compassionate stance adopted by management towards the team. **(Research method – interpretation; People – senior management, three senior designer/programmers, three programmers, HCI designer, knowledge engineer; Project phase – requirements, code design and implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

The extent to which management understood the programming task appeared to have an affect on the project and the team.

Related data samples

SfK's senior management and sales personnel had little first hand knowledge of the technical programming task. **(Research method – participant-observation / ethnographic interview; People – senior management and sales personnel; Project phase – requirements, code design and implementation (early); Data Source – field notes June 1993 – September 1993)**

Misconceptions about the programming task appeared to be held by management **(Research method – participant-observation; People – senior management; Project phase – requirements, code design and implementation (early); Data Source – field notes June 1993 – September 1993)**

Management also expected software architects and programmers to train themselves on new approaches required by the new tools being used on [project 1]. However, the magnitude of transition from structural data driven programming to object-oriented event driven programming (effectively a paradigm shift) was not recognised... **(Research method – participant-observation / ethnographic interview; People – senior management; Project phase – requirements, code design and implementation (early); Data Source – field notes June 1993 – September 1993)**

This caused further problems for the technical core of the team as the prototype had raised expectations of what could be produced in Windows, and they were expected to construct the software for real. The most dangerous of these expectations were held by management who did not fully appreciate the considerable difficulty of the programming tasks involved in the construction of the [project 1] software. **(Research method – participant-observation / interpretation; People – senior management, senior designer/programmers and programmers; Project phase – code design and implementation (early and late); Data Source – field notes June 1993 – September 1993)**

Fortunately, ... the project manager realised the benefits of HCI and my new role on the team. I think in particular, he knew that my concern for the UI was closely aligned with his concerns for the quality of the software. It was because the project manager decided to champion the HCI cause and support my user perspective that I was able to get any changes made at all. **(Research method – participant-observation; People – project manager and HCI designer; Project phase – implementation (early); Data Source – field notes June 1993 – September 1993)**

Management commitment to the integration of HCI considerations into mainstream software development is necessary in order to ensure that HCI issues are heard in a predominantly technically oriented software team. The following data samples illustrate management support for HCI alongside a certain negativity towards programming.

Related data samples

Management attitudes also contributed to attitude problems as programming was often heard to be called, "the easy bit" of the development. Management attitudes towards HCI were generally more flattering. **(Research method – participant-observation; People – senior management; Project phase – code design and implementation (early and late); Data Source – field notes June 1993 – September 1993)**

The job of programming the [project 1] system actually became somewhat of a thankless task as the HCI designer would be recognised for his design efforts and the programmers were merely expected to build the software precisely to the design. If a programmer wanted to excel, it would have to be by working faster - they were not in a position to improve on the design (only deviate from it) and any coding that was clever would go unnoticed by management. **(Research method – participant-observation / interpretation; People – senior management, senior designer/programmers, programmers and HCI designer; Project phase – code design, HCI design and implementation (early and late); Data Source – field notes June 1993 – September 1993)**

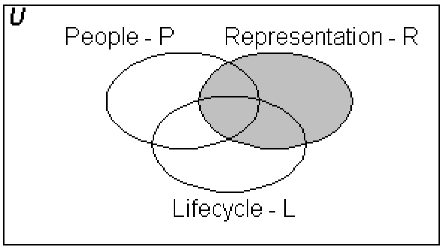
It is also essential that sufficient budget is allocated to HCI activities and that this too has suitable management backing. The following data sample refers to a situation where insufficient budget had been allocated for HCI activities.

Related Data Sample

...the project manager was very keen to know when the ***detailed designs*** would be finished, so that the programmers could get on with their work and my involvement would be over. **(Research method – participant-observation / ethnographic interview; People – project manager and HCI designer; Project phase – HCI design; Data Source – field notes September 1992 – May 1993)**

2.9.1.6.2 Representation Issues

Set, **R**



'Categories of Influences' which can be considered Representation Issues

The Representation Issues set contains 'categories of influence' which relate to representations used in the software production process and their effectiveness at facilitating comprehension.

Software engineering technical notation, e.g. DFD

Technical software engineering notations take many forms within a software development. For example, Data Flow Diagrams (DFDs) were used to model data flows and data processes in the proposed software and were commonly broken down into several levels of detail. The views of the proposed software afforded by such abstract technical representations are apparently not easy for all members the team to understand.

Related data samples

...the representation... (*object model*) ...did not lend itself to quick interpretation...**(Research method – participant-observation; People – two senior designer/programmers, two programmers and HCI designer; Project phase – code design and HCI design; Data Source – field notes June 1993 – September 1993)**

It was felt that the task models should have solved ...(*the problems that team members were having in visualising the project 1 system in use*)... but in their existing format did not provide what was required. **(Research method – participant-observation; People – two senior designer/programmers, two programmers and HCI designer; Project phase – HCI design; Data Source – field notes June 1993 – September 1993)**

Interpretation of many of the representations seemed to require a degree of training and an appreciation of internal design considerations. As there was no time for team members to learn about new notational techniques, team members adapted familiar techniques. Adapting techniques from a structured programming paradigm for an object-oriented paradigm was not always appropriate.

Related data samples

Following an examination of object-oriented design (OOD) representations and techniques …it was evident that the OO paradigm was not easy to learn, nor was it particularly accessible (there were many conveniently simple examples in available texts) **(Research method – participant-observation; People – two senior designer/programmers, two programmers and HCI designer; Project phase – code design and HCI design; Data Source – field notes June 1993 – September 1993)**

Given the time constraints of the project it was felt that designers should use the techniques and representations that they knew well and adapt them for use with an event driven Windows environment. ...**(Research method – participant-observation; People – project manager, two senior designer/programmers, two programmers; Project phase – code design; Data Source – field notes June 1993 – September 1993)**

DFDs... ...proved to be a difficult representation to apply to the architectural design of the [project 1] system. The primary reason for this is that Windows is an event driven environment and DFDs were designed for data driven environments. **(Research method – participant-observation; People – senior designer/programmer; Project phase – code design; Data Source – field notes June 1993 – September 1993)**

(*The inappropriate nature of DFD representations*) ...was dealt with by adding control flows to the diagram... (*but*) ...this complicated the diagram and was perhaps an inappropriate extension to the representation. **(Research method – participant-observation; People – senior designer/programmer; Project phase – code design; Data Source – field notes June 1993 – September 1993)**

It was more likely the familiarity with the DFD notation rather than its effectiveness as a representation which led to its adoption. **(Research method – participant-observation; People – two senior designer/programmers and two programmers; Project phase – code design; Data Source – field notes June 1993 – September 1993)**

…the technical core of the team were familiar with techniques and representations that have been used for design of structured programming architecture for many years but these proved inappropriate for the Windows event driven OO architecture **(Research method – participant-observation; People – two senior designer/programmers and two programmers; Project phase – code design; Data Source – field notes June 1993 – September 1993)**

Although the software engineering notations described are fundamentally rooted in the software production process, the diagrams are often inaccessible to team members, the user and client. Furthermore, the complexity of technical representations (also linked with the skill with which they were constructed) is thought to be a cause of confusion and misinterpretation even between team members from the same discipline.

Software Prototypes

Software prototypes in this context refer to animated visual prototype models showing the concept of the proposed software and HCI design intent. These prototypes had no functionality but presented a façade, giving the illusion of the proposed software's functionality.

These prototypes were developed early on in the production of complex software as a means to help users and clients appreciate the concept of the proposed software (thus enabling their input to alter the direction of the software before any actual development work has occurred). These prototypes also had another purpose: they have been seen to be an effective means of helping team members visualise and comprehend the concept of the proposed software.

Related data samples

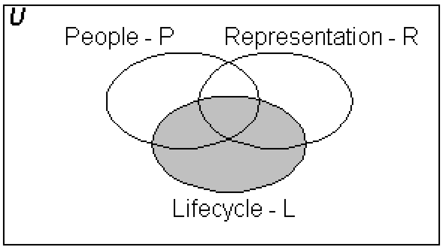
Representation of the [project 1] system from an HCI perspective was eventually conveyed primarily by screen sketches, walk-throughs and visual prototypes. **(Research method – participant-observation; People – three senior designer/programmers, three programmers and HCI designer; Project phase – HCI design; Data Source – field notes June 1993 – September 1993)**

direct integration of HCI and architectural representations was not achieved, rather HCI representations such as sketches or the prototype, offered an alternative view of the [project 1] system than was seen in the OOD or DFD representations. **(Research method – participant-observation; People – three senior designer/programmers, three programmers and HCI designer; Project phase – code design and HCI design; Data Source – field notes June 1993 – September 1993)**

In addition to the written specification, a software prototype was produced to help a variety of people to visualise the complicated results module of the [project 1] system. **(Research method – participant-observation; People – three senior designer/programmers, three programmers and HCI designer; Project phase – code design and HCI design; Data Source – field notes June 1993 – September 1993)**

2.9.1.6.3 Lifecycle Issues

Set, **L**



'Categories of Influences' which can be considered Lifecycle Issues

The Lifecycle Issues set contains 'categories of influence' which relate to the lifecycle used in the software production process. The core 'categories of influence' in this set are in the intersections with other sets and are therefore covered under the intersection heading.

HCI Criticality in the Design Phase

With the introduction of a single HCI designer into a software team, the HCI design task can become a project bottleneck and therefore, a project critical activity. Delays in the HCI design can delay the whole project as programmers and IT designers cannot begin their work until the basic structure of the user interface has been designed. This clearly has an influence on the software production process but it primarily effects the lifecycle and project management.

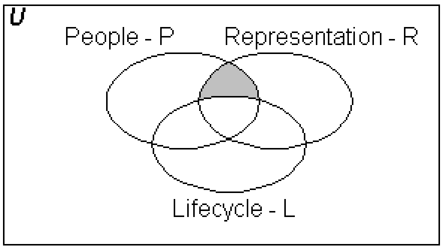
Related data samples

The HCI design activity was on the critical path of the project for a significant period of time during the design phase. **(Research method – participant-observation / ethnographic interview; People – project manager and HCI designer; Project phase – HCI design and implementation (early and late stages); Data Source – field notes June 1993 - September 1993)**

One of the reasons for the integration and raised status of the HCI activity was that it was soon seen to be a bottleneck in the process. Ultimately the HCI detailed design activity had up to five programmers awaiting designs before they could progress. The HCI design activity was also late finishing and directly reduced the amount of time available for programming. **(Research method – participant-observation / ethnographic interview; People – project manager and HCI designer; Project phase – HCI design and implementation (early and late stages); Data Source – field notes June 1993 - September 1993)**

2.9.1.6.4 Intersection of People and Representation issues

Intersection, **(P ∩R) / L**



‘Categories of Influences’ which are both People and Representation issues but which are not related to the lifecycle

This intersection is dominated by the Visualisation ‘category of influence’.

Visualisation

The visualisation ‘category of influence’ is fundamentally related to the effectiveness of representations to convey a ‘visualisation’ (or some form of understanding) and people’s ability to visualise and to articulate their visualisation.

> **Related Data Sample**
>
> The FTM [future task model] proved particularly effective at helping management and sales people and some team members (those with little domain knowledge) visualise exactly how the [project 1] system would integrate with the user’s tasks and how those tasks would be changed. **(Research method – participant-observation / interpretation; People – senior management, sales personnel, two senior designer/programmers, two programmers and HCI designer; Project phase – HCI design; Data Source – field notes June 1993 – September 1993)**

The effectiveness of software engineering technical notations like DFDs are dependent on the reader having sufficient skills and training to be able to visualise from such abstract notations. Not only are such representations often difficult to produce (and there are many varieties of such notations) their use is restricted to a technical audience. Data samples relating to this finding can be found in the software ‘engineering technical notation’ section of 2.9.1.6.2.

Conversely, the software prototype is a representation, which allows a range of diversely skilled people within the software team (and users, clients, etc.), to understand the underlying concepts of the proposed system. It achieves this by enabling them to visualise the proposed end result of the software development (i.e. as a façade of the proposed system) at the early stages of the development process. In addition to the data samples that follow, those appearing under the software prototypes section of 2.9.1.6.2 are relevant to this issue.

Related data samples

As a tool for HCI design, prototyping allowed an experimental approach to design ideas which could be quickly tested and animated in a realistic Windows style of interaction. A benefit of this approach was the aided visualisation of the design that the tool could provide. **(Research method – participant-observation; People – HCI designer; Project phase – HCI design; Data Source – field notes June 1993 – September 1993)**

Many aspects of the required interaction, quality and style of the software are demonstrated in the prototype more effectively than in the written specification. **(Research method – participant-observation; People – senior management, sales personnel, project manager, three senior designer/programmers, three programmers, knowledge engineer and HCI designer; Project phase – HCI design, implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

The effectiveness with which an individual is able to visualise appears to be dependent upon specific characteristics. These include their role, disciplinary background, raw skills, ability, experience and their personal characteristics (or personality). Similarly, team members need to share a vision of what they are producing. This means they need to articulate and communicate complex and abstract concepts within a multi-disciplinary team context. If team members have different interpretations (perhaps due to interdisciplinary differences), this can cause the product to be pulled in different directions causing a breakdown in conceptual integrity. Team issues such as unfamiliarity can also contribute to these difficulties (i.e. a team of people who are not familiar with each other may experience communication problems due to this).

Related data samples

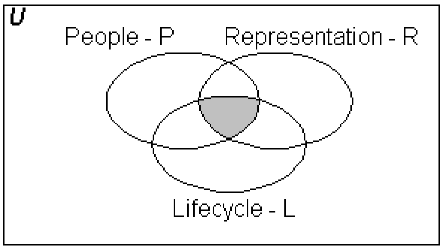
…the effectiveness of accurately conveying the HCI design intent through an animated prototype appeared to depend upon how members of the multi-disciplinary development team interpreted what they saw in the visualisation. **(Research method – interpretation; People – three senior designer/programmers and three programmers; Project phase – HCI design, implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

This lack of any initial building blocks for individuals' mental models of the proposed system complicated team communication and focus, as was seen by the levels of misunderstanding that occurred. **(Research method – interpretation; People – three senior designer/programmers and three programmers; Project phase – code design, HCI design and implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

As designers, software architects and programmers had not seen a system similar to the [project 1] software, initial expectations of what would be produced were vague, and mental models had to be largely constructed from scratch. **(Research method – interpretation; People – three senior designer/programmers and three programmers; Project phase – code design, HCI design and implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

2.9.1.6.5 Intersection of People and Representation and Lifecycle issues

Intersection, **(P ∩R ∩L)**



'Categories of Influences' which are People, Representation and Lifecycle Issues

This intersection of all of the sets in the Venn Diagram contains two 'categories of influence'. They should not be seen as dominant in the model solely by virtue of their appearance in all of the categories. The two categories relate to general project understanding (including domain knowledge and an understanding of project objectives) and changes in requirements.

General project understanding, domain knowledge, project objectives, etc.

The general understanding of the project, domain knowledge, project objectives, etc. held by team members is related to people, lifecycle and representation issues.

During project 1 software development, there was a poor distribution of general project understanding amongst team members. This is thought to be the cause of serious misunderstandings within the team and a number of unnecessarily lengthy project meetings.

Related data samples

The goals and objectives of [project 1] were not clear to all members of the development team. **(Research method – participant-observation / ethnographic interviews; People – two senior designer/programmers and three programmers; Project phase – code design, HCI design and implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

Poor distribution of domain knowledge created a barrier to the formation of a cohesive team understanding of the objectives of the software and its context of use, i.e. there were difficulties in aligning the individuals' mental models. **(Research method – interpretation; People – project manager, three senior designer/programmers, three programmers, knowledge engineer, HCI designer; Project phase – code design, HCI design and implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

The effect of poorly distributed domain knowledge and therefore unaligned mental models was lengthy project meetings (often with no concrete conclusions), many misunderstandings, heated discussions and frustratingly simple explanations to team members who had little domain knowledge. **(Research method – participant-observation; People – project manager, three senior designer/programmers, three programmers, knowledge engineer, HCI designer; Project phase – code design, HCI design and implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

Not unlike visualisation, team members appear to have an inherent ability to understand software concepts, project goals, commercial goals, etc. based on some of the individual level characteristics outlined (e.g. disciplinary background, skills, abilities and personality). It should also be recognised that team members can be unwilling to attempt to gain a general project understanding; they may not consider it part of their role to gain such knowledge or they may consider it irrelevant.

Related Data Sample

Software architects and programmers rarely sought to gain any domain understanding as they saw this beyond the scope of their role **(Research method – participant-observation / ethnographic interview; People – two senior designer/programmers and two programmers; Project phase – code design & implementation (early & late stages); Data Source – field notes June 1993 – September 1993)**

The software lifecycle plans the involvement of team members during the production process. The project plan, based on the lifecycle, is used to manage the limited resources available to a project. Such a project plan may dictate that certain team members cannot join the project until a certain phase has been reached. Common examples of this include introducing programmers halfway through the development process, and reassigning people used in the early requirements elicitation stages to another project. The lifecycle also exerts influence on the order that tasks are carried out, which can effect the integration of HCI design considerations within the process. Therefore, the control of resources, which is defined by the lifecycle (or project plan based on it), has an effect on the level of general project understanding team members can be expected to gain. Thus, a programmer brought on to a project late should not be expected to be able to evaluate the relevance of their programming approach in the general project context, rather they should be expected to follow a tightly specified design.

Related data samples

A Gantt chart style project plan reinforced the waterfall lifecycle approach to the development and served to allocate tasks to individuals. **(Research method – document analysis; Project phase – code design and implementation (early stages); Data Source – field notes September 1992 – May 1993)**

Following the production of the specification document, the [project 1] team grew to around 10 people. **(Research method – participant-observation; Project phase – code design; Data Source – field notes September 1992 – May 1993)**

The effectiveness of representations used to convey general project understanding is a limiting factor when a team is trying to gain a good general project understanding. A complex, disciplinary specific representation may only be effective for some team members, leaving others with a gap in their general understanding (data samples relating to this finding can be found in the software 'engineering technical notation' section of 2.9.1.6.2). An unattractive or poorly written document is another example of a barrier to gaining consistent general project understanding.

Related Data Sample

Each section of the Specification was produced by the relevant disciplinary area. ... Every section in the document was written in each individual's own preferred font and style. The editing of the Specification concentrated on checking the correctness of information in the document rather than presentation and readability. **(Research method – document analysis; Project phase – requirements; Data Source – field notes September 1992 – May 1993)**

Changes in requirements

Changes in requirements have an effect on, or are effected by lifecycle, people and representation issues.

Changes in requirements appear to be unavoidable in a complex software development. There appear to be many reasons for this, but the realities of the commercial software lifecycle are perhaps central to these. Requirements Specification documents are usually written during the preliminary phases of a software development; in fact some such documents form a contract with the client. Unfortunately, during these early stages, of a project it is likely that the requirements engineers and other team members will have had limited exposure to the issues in the Requirements Specification. Therefore, as the project continues and client/user contact increases, it is likely that requirements engineers will gradually gain a deeper understanding of the real requirements. This can mean that software team members become aware of better ways of meeting the client/user requirements, so changes in requirements may occur.

Related data samples

The development culture emphasises the team over the written documentation, which should facilitate changes as team members see fit but the development process and the fact that the specification is a form of contract, conspired against the intended culture. **(Research method – participant-observation/interpretation; People – senior management, project manager, three senior designer/programmers, three programmers, knowledge engineer, HCI designer; Project phase – implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

...the lack of domain knowledge at the outset of the project did not provide a good basis for specifying software requirements or objectives and also lead to changes in requirements throughout the project duration (as domain knowledge improved). **(Research method – participant-observation; People – senior management, project manager, knowledge engineer, HCI designer; Project phase – requirements, HCI design, implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

The lifecycle drives the software project in such a way that changes in requirements are inevitable and have a strong influence on team members. For example, economic considerations associated with the lifecycle dictate that team members are added to a project following certain linear phases. These late-arriving team members cannot hope to understand the background of the project and bring with them new interpretations of the design.

Related Data Sample

Following the production of the specification document, the [project 1] team grew to around 10 people. **(Research method – participant-observation; Project phase – code design; Data Source – field notes September 1992 – May 1993)**

During the initial stages of the project the lifecycle also dictates that it is necessary to estimate the cost of the software. This cost estimation process involves estimating the amount of work (effort) it will take to produce the required software. If requirements change subsequent to the effort estimation, it is likely that additional work will be necessary. Therefore, managers are usually concerned about changes in requirements. Consequently, team members are required to complete their assigned aspect of the work within the original estimated timescales. Thus, changes in requirements can cause friction in the team as team members (especially programmers) often exhibit serious concern over any changes which they perceive may effect them meeting performance targets (i.e. completing the code writing according to initial estimates). Clearly the way that changing requirements are managed also has a potentially strong influence on people within the software production process.

Related data samples

…any changes that were required to a module were fiercely resisted by the technical person responsible for it, fearing that the changes would cause them to exceed their initial estimates. In fact, any changes soon became met by a negative reaction (almost a reflex) on the part of some of the technical people. **(Research method – participant-observation/ethnographic interview; People –two senior designer/programmers, three programmers; Project phase – implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

Increased quality arising from these changes was of minor importance to the technical core of the team, compared with staying within estimated budget. **(Research method – participant-observation/ethnographic interview; People – two senior designer/programmers, three programmers; Project phase – implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

Representation issues influence changes in requirements in two main ways. Firstly, misinterpretation of representations can cause team members to perceive something to be a requirement change when in fact it is just a correct interpretation of the original representation. Clearly this can create tension within the team if the person estimating the amount of work has misinterpreted the design intent shown in the representation (the above data samples illustrate the effects of this). The second influence that representation issues can have is on communicating the changing requirements and updating the team's shared vision of what is being produced. Representing changes in requirements is difficult as many representations of the software may need to be changed at a time when it is inconvenient to do so because team members are already swamped in paperwork. However, these changes must be communicated throughout the team using some form of representation so appended notes are often used to record changes in requirements and subsequent design changes.

Related Data Sample

It should be noted that each team member was usually overloaded with paper (memos, updates, designs, etc.) throughout the process **(Research method – participant-observation/interpretation; People – senior management, project manager, three senior designer/programmers, three programmers, knowledge engineer, HCI designer; Project phase – code design, HCI design, implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

Changes in requirements can cause a serious disturbance to the software production process and the people involved in it. The need to represent changes and to account for misinterpretation also contribute to the realities of this particular problem. Although inevitable, changes in requirements can have a considerable impact on the software development.

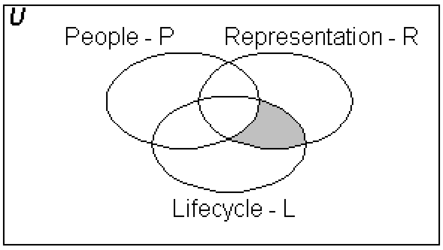
Related data samples

Changes to requirements or 'moving the goal posts', was a significant feature of the [project 1] development. These changes were observed to put a strain on the team and give rise to many misunderstandings throughout the process.**(Research method – participant-observation/interpretation; People – senior management, project manager, three senior designer/programmers, three programmers, knowledge engineer, HCI designer; Project phase – requirements, code design, HCI design, implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

Changes in requirements from the Specification document or from the currently accepted designs proved a considerable disturbance to the development process... **(Research method – participant-observation; People – project manager, three senior designer/programmers, three programmers, knowledge engineer, HCI designer; Project phase – code design, HCI design, implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

2.9.1.6.6 Intersection of Representation and Lifecycle issues

Intersection, **(R ∩L) / P**



'Categories of Influences' which are both Representation and Lifecycle issues but which are not related to People issues

This intersection is dominated by the Specification 'category of influence'.

Specification

The specification 'category of influence' is concerned with the specification document as a representation of the proposed functionality of the software. The timing of the production of the specification document is dictated by the lifecycle.

Specifications are used within the software process to state *what* the proposed software should do and *how* this will be achieved. The specification is written for several reasons and is aimed at a readership ranging from people with no previous computer experience to IT designers with 20 years experience.

Related Data Sample

In addition to the multi-disciplinary nature of the software team, managers and the pseudo-client/user, each with different backgrounds and perspectives needed to understand the specification at some level. **(Research method – participant-observation; People – senior management, project manager, three senior designer/programmers, three programmers, knowledge engineer, HCI designer; Project phase – requirements; Data Source – field notes June 1993 – September 1993)**

The specification is supposed to be clear and detailed enough for it to act as:

- a contract between the software supplier and the client for what will be delivered;

- a representation of the proposed software in a way that the end-user can understand and agree to;

- a key working document for use by the software team, giving sufficiently tightly specified information to guide the next stage of the project.

Related data samples

the Specification was signed-off at the highest level inside SfK and was used by the project manager as a kind of contract to what had been agreed we would produce. **(Research method – participant-observation/interpretation; People – senior management, project manager; Project phase – requirements; Data Source – field notes June 1993 – September 1993)**

The loss of expertise in the team and addition of new members created a burden of communication on the Specification, which had to on one hand, capture lost expertise and on the other, inform the new team members. **(Research method – participant-observation; People – senior designer/programmers, two programmers, knowledge engineer, HCI designer; Project phase – requirements, code design, HCI design, implementation (early stages); Data Source – field notes June 1993 – September 1993)**

This clearly gives rise to some representation difficulties. Further difficulties become apparent when the authoring skills of the specification author are considered. The specification can be written by a number of people (e.g. in project 1 seven people were involved), each with their own writing style. The writing of the specification is not perceived to be a representation task in software development so steps like editing are considered relatively low priority. The usability of a specification can be severely limited by the quality of the authoring and editing and the multiple roles (i.e. contract, system visualisation and technical document) it is asked to play.

Related data samples

…the diversity of uses and users of the specification and the complexity of the authoring task were problematic. **(Research method – participant-observation/interpretation; People – project manager, two senior designer/programmers, one programmer, knowledge engineer, HCI designer; Project phase – requirements; Data Source – field notes June 1993 – September 1993)**

The problem of diversely skilled readership was addressed by using different representations within the specification to give different views on the software. **(Research method – document analysis; Project phase – requirements; Data Source – field notes September 1992 – May 1993)**

Each section of the Specification was produced by the relevant disciplinary area. ... Every section in the document was written in each individual's own preferred font and style. The editing of the Specification concentrated on checking the correctness of information in the document rather than presentation and readability.**(Research method – document analysis; Project phase – requirements; Data Source – field notes September 1992 – May 1993)**

The [project 1] Specification did not perform well in this role [as a working document for the team] due to the poor quality of the document. **(Research method – participant-observation; People – project manager, three senior designer/programmers, three programmers, knowledge engineer, HCI designer; Project phase – HCI design, implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

Many aspects of the required interaction, quality and style of the software are demonstrated in the prototype more effectively than in the written specification. **(Research method – participant-observation; People – senior management, sales personnel, project manager, three senior designer/programmers, three programmers, knowledge engineer and HCI designer; Project phase – HCI design, implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

In the commercial software development lifecycle, completion of the specification is a key milestone in the development of the software. It is also the first key milestone so there is a lot of pressure to produce it quickly. The specification can be completed while there are still considerable unknowns in the future software development path. Such unknowns often include performance issues (e.g. how long a database will take to retrieve a record), integration issues (e.g. are suggested software tools or platforms compatible) or uncertain domain information (e.g. there may be uncertainty in the user task models which have been produced). Therefore, the timing of the specification is an important issue which is dictated by lifecycle issues.

Related data samples

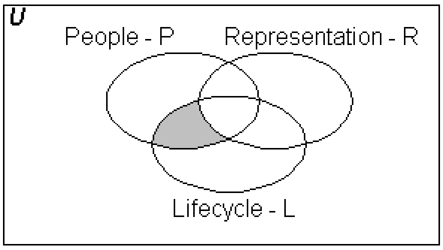
The completion of the specification was such a significant milestone that the project manager presented the team with a big cake (and team-talk) to mark the occasion. **(Research method – participant-observation; People – project manager, three senior designer/programmers, three programmers, knowledge engineer, HCI designer; Project phase – requirements; Data Source – field notes June 1993 – September 1993)**

Whilst emphasis was placed on the Specification document as a key milestone in the project, it was understood that information in the Specification could be questioned and changed. **(Research method – participant-observation; People – project manager, three senior designer/programmers, three programmers, knowledge engineer, HCI designer; Project phase – requirements, code design, HCI design, implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

Producing the Specification was seen by most team members as a chore, particularly as it meant committing themselves to a design that they were as yet unconvinced about. **(Research method – participant-observation; People – project manager, two senior designer/programmers, one programmers, knowledge engineer, HCI designer; Project phase – requirements; Data Source – field notes June 1993 – September 1993)**

2.9.1.6.7 Intersection of People and Lifecycle issues

Intersection, **(P ∩L) / R**



'Categories of Influences' which are both People and Lifecycle issues but which are not related to Representation issues

The intersection which denotes considerations which are primarily People and Lifecycle issues includes 'categories of influence' relating to the changing composition of the software team and emphasis on individual over team approaches.

Changing Composition of Team

The changing composition of the team is related to both People and Lifecycle issues.

The changing composition of the team effects the general level and distribution of understanding held by the team and individuals in it. Knowledgeable team members may leave the team and take knowledge with them and new people may be brought into the team with no project knowledge. An example of this is that requirements engineers who work on the early stages of the project may be taken off the team when the implementation phase begins and programmers may be brought into the team at this time. Maintaining a shared vision (i.e. conceptual integrity) of complex and abstract software concepts is made more difficult by changing personnel.

Related data samples

The main effect of such changes ...(*in the composition of the team*)...was that some knowledge and experience was lost from the team, and new team members (often programmers) needed instruction and training. **(Research method – participant-observation; People – senior designer/programmers, two programmers, knowledge engineer, HCI designer; Project phase – requirements, code design, HCI design, implementation (early stages); Data Source – field notes June 1993 – September 1993)**

The loss of expertise in the team and addition of new members created a burden of communication on the Specification, which had to on one hand, capture lost expertise and on the other, inform the new team members. **(Research method – participant-observation; People – senior designer/programmers, two programmers, knowledge engineer, HCI designer; Project phase – requirements, code design, HCI design, implementation (early stages); Data Source – field notes June 1993 – September 1993)**

This changing composition can be considered a lifecycle issue, strongly driven by commercial needs to optimise the utility of staff resources. Team members, who perform well at the start of the software process, when lots of communication with clients and users is required, are not usually the same people that actually write the code.

Related Data Sample

Resource availability demanded that during the course of the project, the composition of the team and the roles of team members had to change. **(Research method – participant-observation; People – senior designer/programmers, two programmers, knowledge engineer, HCI designer; Project phase – requirements, code design, HCI design, implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

Emphasis on Individual over Team Approaches

The emphasis on individual over team approaches to software development is both a People and a Lifecycle issue.

At some level, writing software code can be considered an individual task as it is usual for programmers to write modules of code on their own. However, much like a joint authoring situation, these individually written modules must ultimately be combined into a coherent whole software application.

Related data samples

For implementation, the specification was divided into parts that could be programmed by an individual. **(Research method – participant-observation; People – project manager, three senior designer/programmers, three programmers; Project phase – requirements, code design, implementation (early stages); Data Source – field notes June 1993 – September 1993)**

When bugs were discovered in the implementation, I was surprised at how the programmers took them personally. On discovery of a bug, some programmers were keen to attribute blame for it, seemingly to clear their own name. In some ways this seemed to emphasise the individualist rather than team approach to the development. **(Research method – participant-observation / interpretation; People – three senior designer/programmers and three programmers; Project phase – implementation (early & late stages); Data Source - field notes June 1993 – September 1993)**

the team oriented approach demanded by SfK to produce the [project 1] software did not suit all of ..(*the members of the technical core of the team*)... many of whom would often comment that they were being asked to do something which was, “not in the Spec”. **(Research method – participant-observation; People – senior designer/programmer and two programmers; Project phase – code design, implementation (early & late stages); Data Source - field notes June 1993 – September 1993)**

It is apparent that even within the programming role, individual programmers have specific strengths and weaknesses that define precisely what kind of code modules they are best able to implement. For example, some programmers are good at user interface coding as they are precise and consider how the module will communicate with the user, whilst others may have a particular skill with syntax which allows them to push the limits of what can be implemented in the programming language. Team skills (i.e. non-technical skills) which some individuals possess are also relied on in a software project. For example, some team members naturally lead others or are particularly good at making sure everybody in the team understands what is required of them.

Related data samples

One programmer had skills which would lead us to view him as a 'hacker' (syntax wizard). I once overhead a conversation between him and the Borland C++ technical support line. He was phoning to find out more information about the features dialog box after it had been made scrollable. The technical support person informed him that it was not possible to make a dialog box scrollable. The programmer had already made it scrollable and was phoning to find out more features of this phenomenon. He had therefore apparently achieved the impossible and was now on his own as far as Borland were concerned. Another characteristic of the same programmer was his 'ability' to have no memory of the previous day. On one occasion, when asked to try something out for a meeting the following week, he tried it but forgot the outcome by the time the meeting came around (and he rarely wrote anything down). **(Research method – participant-observation; People – programmer; Project phase – implementation (early stages); Data Source - field notes June 1993 – September 1993)**

Younger and more inexperienced team members for example, the HCI designer and the knowledge engineer, out of necessity, took on more responsibility in managing the development, based largely on their social skills. **(Research method – participant-observation / interpretation; People – knowledge engineer & HCI designer; Project phase – code design, implementation (early & late stages); Data Source - field notes June 1993 – September 1993)**

Thus, in practical software developments, it is the individuals' particular skills that are a key consideration when building a team or when distributing the work within the team. Although formal software development procedures and lifecycle may be in place, the 'real' software development demands individual performances in order for the team to function.

Related Data Sample

…the [project 1] development team were not a cohesive team oriented unit but rather a collection of individuals working on the same software. **(Research method – participant-observation; People – project manager, three senior designer/programmers, three programmers, knowledge engineer, HCI designer; Project phase – requirements, code design, HCI design, implementation (early and late stages); Data Source – field notes June 1993 – September 1993)**

2.9.1.7 Summary

The abstract representation of the data as the 'Venn Diagram' conceptual model, the 'categories of influence' identified and the description of these constitute the main finding from the ethnographic study of project 1. This finding provided a narrower focus for the ethnographic study of project 2, to be reported next. This is illustrated by the funnel diagrams in sections 2.8.6, 2.8.6.1 and 2.8.6.2.

2.9.2 Focused Observations of Project 2 Guided by the Venn Diagram Conceptual Model Produced Through Project 1 Investigation

This section covers the ‘focused observations’ of the ethnographic study (see section 2.8.6.2 for a further description of this aspect of the methodology). To further focus the ethnography, the Venn Diagram and ‘categories of influence’ developed through the investigation of project 1 were used as a starting point for the study of a further year-long project. This section begins with a brief introduction to project 2, which was carried out at the same software company with largely the same team as project 1. Next, a detailed report of the focused observations is made using the Venn Diagram and ‘categories of influence’ as the framework for the write-up. The two projects are then compared. Finally, comprehension as a key theme underlying most ‘categories of influence’ is focussed upon. Some categories describe the features of the setting and are causes of the comprehension difficulties, and other categories relate to things that are designed to facilitate comprehension. These are suggested as targets for improving comprehension within a team, in particular, between the HCI designer and the programmers.

2.9.2.1 Introduction to Software Project 2

With project 2, the researcher joined essentially the same core software team as participant-observer, performing the role of HCI designer for the project’s duration (approximately one year). Specific changes to the composition of the team from the project 1 were:

- the knowledge engineer was replaced by an existing member of staff, who was familiar with this kind of role;
- one senior designer / programmer left the team and was replaced by an existing member of staff with strong mathematical skills.

A further difference to the team was the close contact that was maintained with a user from the client organisation. This person was not considered a team member but given their level of contact with the team, this point could be argued.

Project 2 was to develop bespoke software for regional water resource scheduling. At the core of the software were complex linear programming algorithms which optimally scheduled water resources across a complex hydraulic network based on demand forecasts, reservoir levels and river flows. The aim of the software was to provide a water company with a means of minimising the risk and cost of operating their water network. The software was essentially a decision-support system.

The rationale for extending the ethnography to this second project is described in section 2.8.6.2. Essentially the second study enabled the focus of the ethnography to be narrowed (continuing down the funnel), whilst at the same time broadening the exploration of the issues by looking at a different project. Finally, the second study removed several biases that may have been a feature of the project 1 study, for example, the researcher's inexperience of the HCI role, team members' unfamiliarity with each other and with the relevant programming languages.

The application domain was quite different (but equally complex); project 2 had a real client and so requirements were more tightly specified; technical aspects of the development were different, with greater use made of prototypes and with software delivery to be achieved on a client's own graphical platform (still under development).

Some characteristics of project 2 illustrate the kind of software venture embarked upon:

- the software had a commercial client;

- this project was also complex and innovative, no software like this was believed to exist in the marketplace;

- SFK's software personnel had a limited understanding of the user domain, which was the regional operation of water networks, by senior operations personnel and field team leaders. In recognition of this limitation, knowledge engineers were again present in the software team;

- At the outset of project 2, the roles, responsibilities and expectations of individuals within the software team had become better defined than they were at the outset of project 1;

- the technical core of the team (the programmers and IT designers) had experience of the software technology to be used on the project, including Microsoft Windows programming, C++ and Object-Oriented design. However, they were unfamiliar with Microsoft C++ having previously used the Borland version. They were also unfamiliar with the graphical environment that the software was to be delivered to work on - the platform was still under development by the client company;

- the project team consisted of between eight and ten people;

- the project budget was approximately £400,000;

- project duration was one year.

These project characteristics set the scene for the software development and the ethnographic participant-observation findings reported in the remainder of section 2.9.2

2.9.2.2 Overview of the Venn Diagram Conceptual Model of Categories of Influence

The detailed report of the focused observations stage of the ethnography was produced using the Venn Diagram and 'categories of influence' as the framework for the write-up. This was done to reflect the focused nature of the ethnographic study of project 2.

The project 2 software was perhaps more centrally placed in the universe of discourse (complex software projects produced in a commercial environment) than the project 1 software. Project 1 had been a speculative undertaking with no external client; project 2 was bespoke software developed for a water company. Complexities of the core of the project 2 software development (an advanced mathematical model) were comparable with those of project 1 (a KBS). Technical software complexities were more dominant in the development of the project 2 software as it was necessary to integrate with a software environment being developed concurrently by the client. Therefore, both project 1 and project 2 software developments were in the same universe of discourse. However, project 1 could be considered on the edge of it (due to its lack of direct client) and project 2 could be considered central to the universe of discourse.

The People, Representation and Lifecycle sets were seen to be sufficient for the recording of the focused ethnographic study of project 2. It was not deemed necessary to add to them.

2.9.2.3 Data Supporting the Focused Observations

Rationale for this kind of coding scheme is described in section 2.9.1.5. Project 2 data samples have been re-coded to report the research method of the inferences; deliberately vague descriptions of the specific people involved; and approximate identification of project phases from which observations and interpretations were drawn. Finally, the main report comprising written-up field notes is identified. The following table illustrates the re-coding scheme used in section 2.9.2.4.

Table 2.4 Re-coding scheme adopted for reporting data samples for project 2

Research method	participant-observation ethnographic interview document analysis interpretation reflection
People	user senior management sales personnel project manager HCI designer (the researcher) software designer / programmer (senior programmer) knowledge engineer programmer
Project phases[5]	requirements code design HCI design implementation (early stages)[6] implementation (late stages)
Source of Data	field notes October 1993 – August 1994

[5] It should be noted that project phases are not strictly linear, for example, the code design and HCI design proceed in parallel

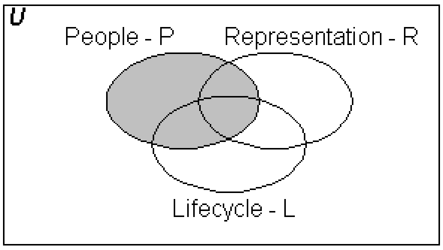
[6] This also encompasses the 'mid' stage of implementation. During early and mid stages of development, the pace of implementation is normal, and various team members are beginning the translation from specification and code design, to implementation.

2.9.2.4 Focused Observations – Detailed Report of Venn Diagram (project 2)

The detailed report that follows utilises the same format as that described in section 2.9.1.6. This report is slightly different from the project 1 report for two reasons. Firstly, the project 1 write-up was intended to describe the Venn Diagram and the categories of influence. The project 2 write-up describes focused observations, using the framework of the Venn Diagram conceptual model, so in general the report is more directly descriptive of the focused observations rather than of the model. Secondly, much of the raw data for project 2 is strictly confidential, which placed further restrictions on the examples that could be included in the report.

2.9.2.4.1 People Issues

Set, **P**



'Categories of Influences' which can be considered People Issues

The sub-levels under the People Issues set correspond to individual level 'categories of influence', team level and management level (see section 2.9.1.6.1 for a further explanation)

<u>Individual-level 'categories of influence'</u>

Personality

During project 2, the client company's own software developments were ongoing. The project 2 software ultimately had to be integrated with the client's software environment, which was itself in its formative stages. It is likely that the reserved and unconfident personalities of some of the key technical individuals within the development team made this integration much more difficult as they shunned liaison with the client company's software developers and therefore created a communication problem. This lack of technical communication meant that the software team's understanding of the client's software environment was also fairly poor.

Technical issues involved in integrating project 2 software and the client's software environment were primarily handled through the development team's requirements engineer and HCI designer due to their people skills, even though these people were not directly involved with writing any code.

Related Data Sample

Once again, one of the programmers had a problem, which he knew none of us could help with but he asked people in our office about it, rather than pick up the telephone to the programmers at the client company. When he returned to his desk I provided encouragement for him to telephone the relevant programmers. On other occasions I have telephoned them myself on a programmer's behalf, having built up a relationship with the client company's programmers it was easy to for me to talk to them. However, this was not ideal as the technical detail of what needed to be said had to explained to me at both ends of the phone. None of the programmers seem to have built up a friendly relationship with the programmers at the client company.**(Research method – participant-observation; People – two senior designer/programmer and two programmers; Project phase – implementation (early stages); Data Source – field notes October 1993 – August 1994)**

Role

Following the experience gained from project 1, the role of the HCI designer was more clearly defined in project 2. Both the HCI designer and other team members had gained a better understanding of the extent, responsibilities and priorities associated with the HCI role. For example, that my skills enabled me to understand software engineering notations and representations was now recognised; previously, notations and representations were considered to be of no relevance to an HCI designer (see section 2.9.2.4.2 – software engineering notation data samples).

Software architects and programmers continued to get along without gaining domain understanding for themselves.

Related Data Samples

Six months into the project and a senior programmer asked me what a blue triangle means [a symbol used in the display]. I can't help feeling exasperated that this person has not yet learned about the most fundamental aspect of the system we are building – the blue triangle represented a reservoir, possibly the most important aspect of the software we were building. How can a senior programmer consider such knowledge beyond the scope of their role when they are making day-to-day decisions that shape the software? **(Research method – participant-observation; People – senior designer/programmer; Project phase – implementation (early stages); Data Source – field notes October 1993 – August 1994)**

Conflicts between role priorities were evident in the project 2 development (see the data sample described in the following section).

Disciplinary Background

At times the fundamental stance adopted by some team members over particular issues apparently related back to their core disciplinary background. The implementation of modeless dialog boxes (i.e. the facility to allow the user to interact with several dialog boxes on screen concurrently, as opposed to the default modal behaviour which dictates that only one dialog at a time could appear on screen) was one example.

Related data sample

Because part of the user's task involved comparing reservoir levels (shown as graphs) in several reservoirs at once, it was necessary to view this information on screen concurrently. To achieve this end the HCI design specified modeless dialogs for the display of reservoir levels. As the development progressed it became apparent that this aspect of the HCI design was being ignored. Over the period of several weeks I would have regular heated discussions with several key programmers about why this had not been done. They viewed this aspect of the design as an embellishment and would often retort, "what would you rather have, modeless dialogs or software that works?" These programmers team members thought that as long as the information was available to the user in some form (i.e. sequentially) there was no need to expend extra programming effort to enable them to access information concurrently. These team members refused to acknowledge the importance of the user's comparison task. **(Research method – participant-observation/ethnographic interview; People – two senior designer/programmers, two programmers and HCI designer; Project phase – implementation (early and late stages); Data Source – field notes October 1993 – August 1994)**

Skills, ability, competence and experience

At the outset of project 2, technical team members were more familiar with C++, Windows and Object-Oriented Design (OOD), but they were still not experts. Further technical unknowns arose due to the switch to Microsoft Visual C++ (programmers only had experience of Borland C++) and the need to integrate with the client's software environment. Despite the unknowns, the team were more familiar with the tools they would have to use, than they were at the outset of project 1.

At the outset of project 2, the HCI designer had gained considerably more experience through project 1 and other smaller projects which ran concurrently.

It became further apparent throughout project 2 that a person's ability to visualise software concepts is a personal skill or ability. The requirements engineer and the HCI designer were both able to visualise the proposed software effectively, whilst most other team members were not.

Related Data Sample

During demonstrations of the visual prototype, some programmers appear to be limited to contemplation of individual aspects of the system, rather than building an overall conceptual model of it. Such team members regularly came back to ask further questions about the design. Through this questioning, it was apparent that they had little overall conceptualisation of how the system fitted together. **(Research method – participant-observation and interpretation; People – two senior designer/programmers, two programmers and HCI designer; Project phase – code design, implementation (early and late stages); Data Source – field notes October 1993 – August 1994)**

Team-level ‘categories of influence’

Unfamiliarity

The team involved in producing the project 2 software was largely the same team that produced the project 1 software. These people were by now more familiar with each other and this appeared to make things run more smoothly. This was especially noticeable with respect to the acceptance of the HCI designer role, which by now most people understood.

> **Related Data Sample**
>
> Following the introduction of a new team member I am forced to reflect upon how the rest of the team have now gained an acceptance of HCI design. The new team member refused to acknowledge HCI design considerations. This was something I had grown used to, more surprising was that I was no longer fighting this attitude on a daily basis with the rest of the team, as had been the case throughout much of project 1. This apparent change of attitude could have come from their prolonged exposure to the HCI designer role, or conceivably, from the fact that I was now a familiar team member. **(Research method – participant-observation / ethnographic interview; People – three senior designer/programmers, three programmers; Project phase – implementation (early stages); Data Source – field notes October 1993 – August 1994)**

Interdisciplinary Issues

The disagreements about the inclusion of modeless dialog boxes in the software (discussed under the heading ‘Disciplinary Background’ above) could also be labelled an interdisciplinary issue.

A further interdisciplinary issue was that team members from different disciplines appeared to draw different information from a visual prototype according to their disciplinary (or role) priorities (this is discussed further under the ‘Software Prototypes’ section).

Management-level ‘categories of influence’

Management Influence

Project 2 was a very high profile and commercially important software development for the software development company. This led to considerable interest and influence from senior management and the deadline for the software delivery was a more keenly felt commercial deadline than that of project 1.

Senior management had a continuous involvement with the client company throughout the duration of the project. Some of this took the form of knowledge elicitation for the project and gaining greater domain familiarisation. However, the majority of the client contact took the form of consultancy work, aiding the re-organisation of the client company processes. It was not always clear to members of the development team what management were doing at the client company. Feedback of information relating directly to the project was often ineffective.

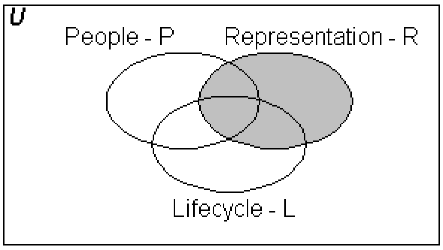
Related Data Sample

Towards the end of project 2 the senior manager of SFK demanded a demonstration of the nearly completed software. This manager had not been involved in the day-to-day activities of the project team and had instead spent the majority of time based at the client company, analysing requirements and becoming involved in business process re-engineering activities (BPR). During this demonstration, in front of the whole team, the manager became convinced that the software had failed to meet her expectations regarding a particular requirement. Consequently, the whole team was extensively hollered at by the manager. This illustrates the lack of communication between the manager that spent a considerable amount of time at the client company, and the software development team. The manager had also failed to take into consideration the effect of her actions on the team that had worked solidly for over a month (including evenings and weekends) to get the software to this stage of development. **(Research method – participant-observation; People – senior management, project manager, three senior designer/programmers, three programmers, knowledge engineer and HCI designer; Project phase – implementation (late stages); Data Source – field notes October 1993 – August 1994)**

Management clearly had an influence on the team, not least relating to team members' morale.

2.9.2.4.2 Representation Issues

Set, **R**



'Categories of Influences' which can be considered Representation Issues

Software engineering technical notation, e.g. DFD

The initial representation of the structure of the project 2 software consisted of an overview of the software, using an *ad hoc* and non-formal representation (conceivably a context diagram). This overview of the software was then turned into an object-oriented design in the form of a class diagram. The main differences between these two forms of representation were that the latter required knowledge of the notation to interpret, was more formal and less ambiguous.

The next level of design detail was represented using Data Flow Diagrams (DFDs). These provided a notation and level of detail which were not easy to interpret and not accessible to people unfamiliar with the notation and other software engineering concepts. DFDs did not provide a suitable representation for object-oriented software development under Microsoft Windows, but they were the representation that most team members were familiar with.

Alternative representations for the detailed design were considered during the design phase but were inaccessible without training and would therefore have involved considerable learning curves.

The detailed software design represented by the DFDs was validated in design reviews by walking-through various functional processes shown in the diagrams.

Related Data Sample

I was invited to several meetings to confirm that these representations matched my conceptualisation of the proposed form of the project 2 software. Because I understood DFD notation, I was able to review (walk through) this representation of the design to ensure that it was consistent with my conceptual model of the design. **(Research method – participant-observation; People – project manager, two senior designer/programmers, knowledge engineer and HCI designer; Project phase – code design; Data Source – field notes October 1993 – August 1994)**

Therefore, this means of reviewing the software engineering representation of the design relied on the HCI designer having a good understanding of the DFD notation and a good mental model of the proposed software.

The primary (most commonly referred to and most accurate) representation of the vision of the project 2 software was the extensive software prototype that was constructed.

Software Prototypes

Extensive use was made of prototypes in the analysis of current user tasks, the modelling of future tasks and as a clear visual representation of the proposed software. Scenarios, derived from task analysis, were built-in to the prototype's façade to facilitate a realistic explanation of the proposed software in context. This scenario-driven approach was found to be particularly effective for helping team members and users visualise how the software would be used.

The prototype was of particular use in illustrating the requirements specification, a representation of the software which several people (including the user) were unable to make sense of. So where the written specification had failed to elicit feedback, the prototype was usually more successful. In some respects, the prototype became a kind of animated specification.

The visual prototype was found to be an inherently useful tool in eliciting requirements from end users.

Related Data Sample

The first time we tried to present the visual prototype to the user we failed to install it on the user's computer. The proposed software was instead illustrated to the user by making use of a complete set of screen printouts from the prototype. Presentation of the proposed software in this way was extremely ineffective and feedback generated was poor, primarily because the user appeared disinterested and unable to comprehend the significance of the screen printouts. A week later I returned and successfully installed the computer prototype on the user's own desktop computer. The difference in the attitude of the user was marked, he was immediately interested, even captivated with what was on screen. **(Research method – participant-observation; People – knowledge engineer, HCI designer and user; Project phase – requirements; Data Source – field notes October 1993 – August 1994)**

The difference in the effectiveness of the prototype demonstration and the screen printout demonstration make this observation particularly noteworthy because the information shown in both presentations was identical. There are several possible explanations for this. One explanation might be that the user was a novice computer user which may have effected his perception of what appeared on the computer screen (the first prototype iteration was little more than an animated slide show). Possibly the best explanation is that the conceptual leap from printed paper to computer screen was too much for this user to comprehend but actually seeing an animated screen presentation on his computer screen made the whole demonstration less abstract. The prototype seemed to empower the user to visualise the software and enable him to provide effective feedback.

When interviewed at the end of the project, the end user had no recollection of ever having seen the prototype that was produced early on in the development process. Even when shown the old prototype side by side with the finalised project 2 software the user still could not recognise it. One possible explanation for this is that the user assumed that the prototype iterations were early versions of the project 2 software. From a software developer's perspective, this is quite hard to understand as the prototype was very clearly non-functional software and very different from early versions of the actual software. However, this is perhaps further evidence of the potential of the prototyping approach for facilitating comprehension and visualisation of software under construction. Such issues raised some concerns around managing expectations, as after only a month of development time the user apparently assumed that most of the software development work was completed.

One problem with the prototyping approach was with the iterative nature of the process. It was apparent that the best feedback was from the end-user and the client, through the use of successive prototype iteration. Because every visit to the client company derived more information, it was difficult to decide when enough had been obtained. Similarly it was difficult to decide when the design shown in the prototype was sufficiently complete and correct. Consequently, the prototyping to continued long into the project, which had the effect of delaying the commencement of programming. In essence, the problem is how to decide when to stop prototype iteration.

During the process of developing the prototype, the client organisation was developing abstract schematic representations (analogous to the London Underground map - pseudo geographical and logically accurate) of their main water resources, treatment and transfers. The identified user of project 2 software was also interested in the design of the schematic representations and would often focus on what were only sample representations of these in the prototype. This contributed to what appeared to be a more general problem with presenting prototypes - what to do when the user wishes to stop the flow of the presentation to discuss an on-screen detail. Of course, this was a major route to generating feedback, but it had a damaging effect on the continuity of a demonstration. It became an art form to focus the user's attention on the particular aspect of the prototype on which feedback was wanted.

Extensive use was made of the prototype in explaining the functionality and HCI design intent of the proposed software to other team members (much more so than was the case in project 1).

The focusing of attention on aspects of the prototype outside the sphere of the current discussion also applied when presenting it to members of the software team. Team members sometimes interpreted what was shown in the prototype in very different ways (this issue is of relevance to 'Interdisciplinary Issues', 'Disciplinary Background' and 'Visualisation' in the conceptual model). The most likely explanation for this appeared to be that people would look at the prototype primarily from the perspective of what it meant to their specific job. For example, a database designer would focus on the data displayed in a dialog box or a user interface programmer may wonder how dialog boxes which can be viewed concurrently would be implemented and if this is really necessary. This proved to be a paradoxical problem as often dummy data had to be displayed in the prototype to provide a suitable context for illustrating other points, but the dummy data itself would detract from the main point.

Utilisation of the prototype as a means of explaining design intent within the team also had some other drawbacks. The programmers exhibited a reluctance to refer to the prototype themselves, instead preferring to ask the HCI designer questions about the design. This caused considerable disruption to the HCI designer's work.

Related data samples

Programmers rarely turned to the visual prototype to ascertain how an aspect of the proposed software had been designed, even though the prototype was available to all of them on the network. Instead, I became the focal point for any queries they had about aspects of the design. Often a programmer would approach my desk with a question that they would invariably require an immediate answer to. Often, I would answer the question by calling up the prototype and demonstrating the feature that the query related to. Quite often the programmer would query an aspect of the design presented and I would explain its purpose and design rationale from my own understanding (this usually related back to the user's tasks). **(Research method – participant-observation; People – three senior designer/programmers, three programmers and HCI designer; Project phase – implementation (early and late stages); Data Source – field notes October 1993 – August 1994)**

One programmer has a peculiar mannerism when I use the prototype to explain the answer to his design questions. He would stand behind me and only half-look at the screen, he would look at the screen in fleeting glances and look away quickly. This mode of interaction persisted with this programmer throughout the implementation of the project 2 software. **(Research method – participant-observation; People – senior designer/programmer and HCI designer; Project phase – implementation (early and late stages); Data Source – field notes October 1993 – August 1994)**

Perhaps some programmers found it easier and more efficient to ask the HCI designer about an aspect of the HCI design. However, others seemed to have a deeper-rooted reluctance to refer to the prototype. As an ethnographer, my interpretation of the behaviour of the programmer that would only half-look at the prototype is that he perhaps felt threatened by the functional requirements and features shown in the prototype and the high expectations which it generated. This may also be part of the explanation for the other programmers asking me questions directly rather than referring to the prototype. With Visual Basic it was often easy to dummy up or imply functionality which would be extremely difficult to construct in practice. The prototype and prototype developer (the HCI designer) received a lot of credit from management for producing the prototype which the technical team members were then expected to produce.

An important flaw in the use of the visual prototype was that design rationale was not conveyed alongside the HCI design. Thus, technical team members were not in a position to negotiate effectively over an aspect of the implementation. It was not obvious to them whether the aspect was fundamental to the design or merely added by the HCI designer on a whim. Because programmers were not in a position to negotiate, the HCI designer needed to have a good appreciation of the implementation language in order to ensure that the design could be implemented. This meant that the programmers were not in a position to suggest improvements to the HCI design because they were not told what each aspect of functionality was trying to achieve and why. This reduced the scope for programmers to suggest neater technical solutions to aspects of the design shown in the prototype. These could have been equally acceptable in relation to HCI design intent and may save several weeks programming effort.

A spin-off benefit of having produced a prototype was that it was frequently used to give demonstrations of how the software would work when completed to people at all levels in the client company. This was sometimes done to demonstrate progress, or to illustrate aspects of the requirements specification, or to show the software to a wider audience (the implications of using the software were far reaching), or to focus the direction of other projects which were to be integrated with project 2 software and advise the client on their IT strategy. Without fail, the prototype was well received at all levels, mainly because it enabled people to visualise how the software would eventually fit in with working practices. For example, it had been difficult to get feedback on the written requirements specification from some quarters, but using the prototype to illustrate it and help people to visualise what the software would do, feedback was forthcoming.

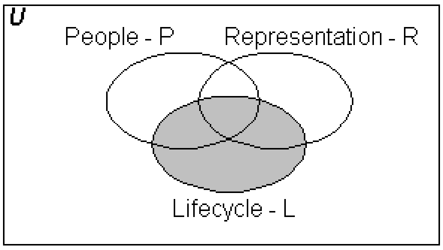
Related Data Sample

I have had to spend several days making the operation of the prototype more obvious so that a manager could operate the façade at a client presentation. Up until now the prototype has been a tool for me to use to gather user requirements and to report these back to the development team. As such, it was operated by various hidden means like mouse clicks in hidden areas of the screen (to avoid adding visual features to the screen that would detract from the system concept being demonstrated). **(Research method – participant-observation; People – senior management and HCI designer; Project phase – requirements; Data Source – field notes October 1993 – August 1994)**

The hidden operation of aspects of the prototype could be a further explanation for programmers reluctance to use it.

2.9.2.4.3 Lifecycle Issues

Set, **L**



'Categories of Influences' which can be considered Lifecycle Issues

HCI Criticality in the Design Phase

As with project 1, the HCI design phase for project 2 again delayed the commencement of programming.

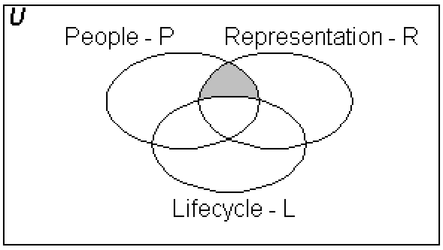
Related Data Sample

I was unwilling to pass over parts of the design on to programmers on a piecemeal basis, in case of the need for significant changes to be made later. **(Research method – reflection; People – HCI designer; Project phase – HCI design and early implementation; Data Source – field notes October 1993 – August 1994)**

Iterations of the prototype and uncertainty of both user tasks and the correctness of the design motivated this attitude. The programming work was therefore delayed and the program possibly suffered as a result. In fact, several members of the team worked a seven day week and a minimum 12 hour day during the month prior to software delivery. This was not the fault of the HCI design activity, more a statement of the amount of domain knowledge and task analysis that was required in order to get the design of the software right.

2.9.2.4.4 Intersection of People and Representation issues

Intersection, **(P ∩R) / L**



'Categories of Influences' which are both People and Representation issues but which are not related to the Lifecycle

Visualisation

The prototype proved to be a powerful means of aiding team members' visualisation of the software (this 'Visualisation' section is closely related with much of the discussion in the 'Software Prototypes' section). The prototype for project 2 spanned the whole system and could be demonstrated with simulated real use scenarios. Prototypes produced for project 1 only covered several individual aspects of the interface and these were less powerful (and less convincing). Overall, it is believed that there were less misunderstandings during the project 2 development than during the project 1 development. This was perhaps due to the early availability of the extensive HCI prototype, which afforded team members a means to visualise how the proposed software should look and behave.

Related Data Sample

A new young programmer joined the team and began asking some tricky questions of the design, the answers to which were not immediately obvious. A meeting was convened to discuss the concerns of this programmer. Such meetings had been commonplace during the project 1 development and were often lengthy and ended without satisfactory conclusion. By contrast, I was surprised that within 5 minutes the meeting resolved the tricky questions. I believe this was because the combined conceptualisation held by the other team members was relatively coherent and consistent, such that new problems arising could not shake the foundations of the jointly held conceptual model. **(Research method – participant-observation and interpretation; People – project manager, three senior designer/programmers, three programmers, knowledge engineer, and HCI designer; Project phase – implementation (early stages); Data Source – field notes October 1993 – August 1994)**

The detailed technical design of the project 2 software was represented by DFDs. The design was verified against the overall design intent reflected in the prototype by the HCI designer and requirements engineer in a design review meeting with the IT designers who had produced the DFDs (see section 2.9.2.4.2 on software engineering notation). Both the HCI designer and requirements engineer had formed an apparently consistent mental model, or vision, of how the project 2 software would ultimately work, through the visualisation afforded by the prototype. Their mental models and understanding of DFDs and other software concepts allowed them to assess whether the software represented in the DFDs would be consistent with the design intent reflected in the prototype. In other words, the HCI designer and requirements engineer were checking that the IT designers had understood the problem and had produced an appropriate design, which would allow the software to function according to the visualisation afforded by the prototype.

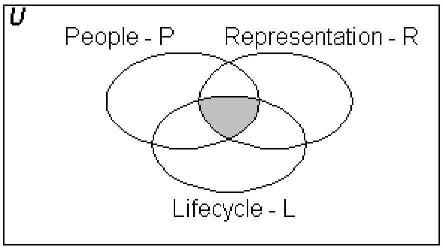
The specification document for the software was primarily aimed at satisfying the clients' requirements for such a document. It was not very effective at facilitating communication and comprehension within the team or the client organisation. The prototype came to be regarded as an animated visualisation of the specification of the project 2 software.

Related Data Sample

One programmer seemed determined to find fault with the specification, complaining that a detailed breakdown of particular aspects of functionality were, "not in the spec". This programmer would not acknowledge the impossibility of specifying every detail of a complex Windows program, and rather than seeking to discover the missing information from another source, would claim that he could not proceed at his task until the missing information was forthcoming. The same programmer would respond slightly more positively to the idea that the prototype showed the design intent, rather than the letter of the detail. For this, and similar reasons, it was decreed that the actual specification of the software comprised the prototype and the written specification document together. **(Research method – participant-observation; People – programmer; Project phase – implementation (early and late stages); Data Source – field notes October 1993 – August 1994)**

2.9.2.4.5 Intersection of People and Representation and Lifecycle issues

Intersection, **(P ∩R ∩L)**



'Categories of Influences' which are People, Representation and Lifecycle Issues

General project understanding, domain knowledge, project objectives, etc.

Several team members developed considerable understanding of the domain, project objectives, etc. However, some team members, notably those making day to day technical decisions which actually shaped the software (at a detailed level), gained little general project understanding and very little domain knowledge. These technical team members often relied upon second-hand information from other sources rather than formulating an understanding of the project and domain for themselves. (see data sample in section 2.9.2.4.1 under role).

Related Data Sample

Poorly distributed domain understanding created the need for a highly detailed specification, which was not effectively facilitated by SfK's development approach. **(Research method – participant-observation / ethnographic interview; People – project manager, three senior designer/programmers, three programmers, knowledge engineer and HCI designer; Project phase – code design, implementation (early stages); Data Source – field notes October 1993 – August 1994)**

Some technical team members insulated themselves from the problems associated with the real world domain of the software and preferred to have their work tightly specified by others (see data sample in the Visualisation category in section 2.9.2.4.4). This created difficulties in negotiations over functionality. Sometimes an aspect of functionality which was of fundamental importance to the user's task would be viewed as an embellishment, rather than a necessity, by the programmers. An example of this was the implementation of modeless dialogs (a technically difficult aspect of the program). Technical team members would often say something like "what would you rather have, software that runs or modeless dialogs?" (see 'Disciplinary Background' in section 2.9.2.4.1).

Changes in requirements

The project 2 software was designed to facilitate a task that was not performed currently (mathematical optimisation of weekly water resources). Developing software for a complex new application made changes in requirements inevitable.

Assessing iterations of the prototype with the user would often generate new ideas or destroy old ones. Some team members perceived this as changing requirements.

Early on in the process, this was an unwarranted criticism as formative inputs were being sought by such sessions. However, continuing the HCI prototyping until a late stage did allow some new ideas (or changes) to be generated late in the development process.

Related Data Sample

Programmers constantly pin the blame for changes to requirements on me. Partly this stems from the fact that my visits with the user do cause requirements to change as I gain a better understanding of the tasks. However, as the HCI designer I am also the first person that has to deal with ANY requirements change, as I have to accommodate the change in the HCI design. Therefore, I am invariably the person that has to break the news of the change to the programmers. Consequently, they believe that all changes are created by me – however much I try and explain otherwise. **(Research method – participant-observation and reflection; People – three senior designer/programmers, three programmers and HCI designer; Project phase – implementation (early and late stages); Data Source – field notes October 1993 – August 1994)**

The complexity of the mathematical formulation of the water network and the linear programming solver in itself generated some changes in requirements at a late stage of development as technical details became finalised. From a mathematical viewpoint, it became obvious that the user would sometimes have to relax the constraints on the linear programming formulation in order to produce a solution. This made it necessary for the HCI Designer to come up with a way of communicating this complex feature to a user who was largely ignorant of the workings of linear programming. In other words, the mathematical model was the source of new requirements on the user interface at a late stage in the development.

The influence of the Lifecycle on programmers' attitudes to changing requirements was the same as for project 1. Programmers were asked to keep within their estimates made at the early stages of the project. Estimates made at such an early stage would almost certainly be incorrect for the development of innovative software. Despite this, programmers tried fiercely to stay within their estimates.

Related Data Sample

On one occasion, it was decided that a part of the system should not be implemented as it would take too long and had little added value. When I broke the news of this to a senior programmer/architect he became quite distressed, sucked his teeth and mumbled that he would have to, "revise his estimates". The implication being that this constituted a change in requirements and as such would inevitably result in more work than sticking to the original plan. Amongst other team members it was a source of considerable humour that this programmer believed it would take longer to do less work. **(Research method – participant-observation; People – senior designer/programmer; Project phase – implementation (late stages); Data Source – field notes October 1993 – August 1994)**

Changing requirements were difficult to communicate to the project 2 team as they were already swamped with paper. It was particularly difficult to keep them informed of changes to the specification.

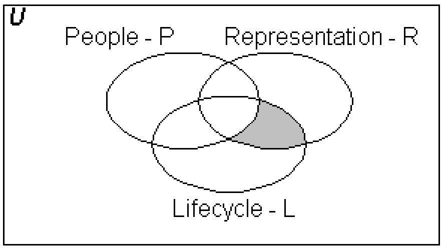
Related data samples

Team members were ultimately provided with around 7 versions of the specification. Some of these contained a front sheet explaining the changes from the previous version and some did not. This didn't seem to matter too much because a number of programmers apparently never referred to the specification and certainly had trouble finding a copy when they needed one. **(Research method – participant-observation and interpretation; People – senior designer/programmer, two programmers; Project phase – implementation (early and late stages); Data Source – field notes October 1993 – August 1994)**

The knowledge engineer undertook to maintain a data dictionary [a list of data items that formed part of the software under construction and a description of each] in a word processor file. The task of keeping this up-to-date and communicating changes to the rest of the team became a significant effort. **(Research method – participant-observation; People – knowledge engineer; Project phase – code design, implementation (early and late stages); Data Source – field notes October 1993 – August 1994)**

2.9.2.4.6 Intersection of Representation and Lifecycle issues

Intersection, **$(R \cap L) / P$**



'Categories of Influences' which are both Representation and Lifecycle issues but which are not related to People issues

Specification

The specification was primarily written for the client to form a contract for the software. It was used at all levels within the client organisation to explain to people (including the user) what the software would do, thus helping them to provide technical input and critique. For some readers in the client organisation, the specification did not effectively fulfil this role. This was apparent, because individuals with strong vested interest in the software who provided little input when they read the specification, provided considerable input when they were shown the prototype (which reflected much of what was stated in the specification).

The prototype regularly performed the role of 'animated specification', for showing a variety of people, e.g. users, team members or other members of the client organisation, what the proposed system would do and how. In fact, it was decided that the prototype formed part of the specification of the project 2 software, but in reality it was treated as a guide by team members (the written specification by contrast was treated as a binding contract).

Related data samples

Ultimately, the use of the specification as a contract proved to be a serious flaw in the flexible approach taken to specification. **(Research method – participant-observation; People – project manager, three senior designer/programmers, three programmers, knowledge engineer, and HCI designer; Project phase – requirements, code design, HCI design, implementation (early and late stages); Data Source – field notes October 1993 – August 1994)**

…as the Specification document was signed-off, it was in the project manager's interests to ensure that no further changes were made as the Specification formed a contract outlining what would be produced by the venture.**(Research method – participant-observation; People – senior management, project manager; Project phase – requirements; Data Source – field notes October 1993 – August 1994)**

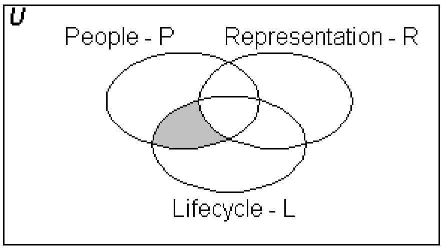
The specification was also used as the main working document for the development team. This use of the specification was clearly compromised by its other use as a client document, from the style it was written in to the information it contained. In addition, there was a tendency (and partly a requirement) to divide the software development into chunks that could be implemented by an individual working alone. Thus, the specification still had a dominant role and encouraged team members to buy-in to chunks of the development, rather than the whole project, despite the emphasis on teamwork and communication being greater on this project than on project 1.

Related Data Sample

The format of the specification was dictated by the client organisation. Our desire to include task analysis as a section in the document led us to bend the rules and slip this section in. The client noticed this, queried it and asked us to remove it from the specification.**(Research method – participant-observation; People – project manager and HCI designer; Project phase – requirements; Data Source – field notes October 1993 – August 1994)**

2.9.2.4.7 Intersection of People and Lifecycle issues

Intersection, **(P ∩L) / R**



'Categories of Influences' which are both People and Lifecycle issues but which are not related to Representation issues

Changing Composition of Team

Nobody left the project 2 development team during the project, although some technical people (mainly programmers) joined the team six months after the project had started. Nobody leaving the team was an unusual occurrence, perhaps due to the commercial significance of the project to the SFK. This may be one of the reasons why the development of the project 2 software progressed smoothly with few misunderstandings and little re-work. A further reason could be that two particular team members (a requirements analyst and the HCI designer) stayed with the team throughout the development. Both of these team members appeared to be good at understanding requirements, conceptualising the software, and conveying this to other team members.

The team apparently maintained a good degree of shared vision of the software (see data sample relating to Visualisation in section 2.9.2.4.4). However, several team members cared little for an understanding of the application domain context of the software (see data sample relating to Role in section 2.9.2.4.1) and had little general understanding of the project objectives. Such team members, however, did have a voice in design meetings, which made technical decisions about the direction of the software, and they also made other day-to-day technical decisions in the course of their own work. These team members seemed reluctant to 'buy-in' to the project as a whole and would be seen to 'buy-in' only to the sections of the software which they produced.

Emphasis on Individual over Team Approaches

Emphasis on the skills of the individual was an important aspect of the project 2 development. During project 2, the skills of one particular programmer tacitly made him the 'Windows Expert'. Ultimately, there was strong emphasis placed upon the skills of this person to bring the software together. It was clear from very early on in the project that much of the development work would fall on this team member (who was an enormously disorganised person with an extremely unstructured approach to the production of software but who was brilliant with syntax - a 'hacker').

Related Data Sample

I took the project manager aside at the outset of the project and pointed out that the development would all come down to one programmer (the Windows expert). Almost a year later he well remembered my prophecy and appeared amazed that I was able to read the future with such accuracy. **(Research method – participant-observation / ethnographic interview; People – project manager, and HCI designer; Project phase – requirements/implementation (late stages); Data Source – field notes October 1993 – August 1994)**

This is only one example from project 2 where the skills of a particular individual were vital to the success of the project. Other examples include the fact that a few team members 'championed' the project, providing drive and initiative to the development. The efforts of individuals outside their general role titles appeared to be vital to the production of the project 2 software.

Related Data Sample

The knowledge engineer and myself have apparently become a kind of knowledge-base for the team. Together we have clear understanding of what the software should do and its context of use, so we get to answer a vast range of different sorts of questions for the rest of the team. Sometimes questions can be quickly answered by either of us based on our conceptualisation and other times we need to put our heads together to come up with the appropriate decision. Occasionally, a question arises that is not covered by our conceptualisations and so needs some research to discover an answer. **(Research method – participant-observation; People – knowledge engineer and HCI designer; Project phase – code design, HCI design, implementation (early stages and late stages); Data Source – field notes October 1993 – August 1994)**

From observation of the project 2 development, it is again clear that the commercial development process necessitates splitting the specification document into chunks to be produced by individual programmers. Thus, a large part of the software production task is carried out by individual efforts, which are later combined.

2.9.2.4.8 Summary

The detailed report of project 2 observations focused by the Venn Diagram and categories of influence is only the beginning of the 'focused observations' section of the funnel. The next stage down the funnel involves a basic comparison of the findings from project 1 and project 2.

2.9.2.5 Basic Comparison of Project 1 and Project 2 Software Developments

A formal comparison of the two case studies would be inappropriate (and was not planned) as there are too many extraneous factors, which contributed to the differences in the software developments. However, as a participant in both software developments, I observed several differences between them. The remainder of this section is my interpretation of these differences.

The most notable difference was that project 2 progressed much more smoothly than project 1. The project 2 team maintained a consistent impression of the software under construction and major misunderstandings (which had been a feature of project 1, often requiring re-work) were reduced. There are several possible reasons for this, which may be considered:

- more use was made of visual prototypes to explain the proposed software to team members. This method of helping the team members visualise and understand probably also helped them maintain a consistent mental model of the software;

- one person who joined the project 2 team and was not a part of the project 1 team, and another team member, who had been on the project 1 team, were apparently strong conceptualisers. These team members were also good communicators who made a considerable effort to explain elements of the design to other team members (who were less able and willing to conceptualise);

- the team members had become more familiar with each other and were able to understand each other better;

- the technical core of the team had become more familiar with implementation languages (C++) and programming under Windows. However, there were still unknowns in this area; Microsoft C++ was used instead of Borland (different function libraries created a learning curve) and the software was to run under a graphical schematic interface (Schematics) which was still under heavy development;

- there was a real client for the project 2 software so requirements were more clearly defined.

Although further comparisons between the projects could be made, this was not the intention of the study and such views cannot be firmly supported.

2.9.2.6 Underlying Finding of the Ethnographic Investigation - Comprehension

Peer analysts rejected the direct association of the 'categories of influence' with specific underlying themes of communication and comprehension, during the analysis of raw data relating to project 1. However, during the ethnographic study of project 2, it is apparent to the researcher that most of the 'categories of influence' in the model have a comprehension dimension. Figure 2.7 illustrates those categories that are related to the issue of comprehension. Some ‘categories of influence’ aimed to convey an understanding of the software being produced, e.g. the specification, and other categories sought to explain why comprehension problems may exist within a software development team, e.g. changing composition of the team.

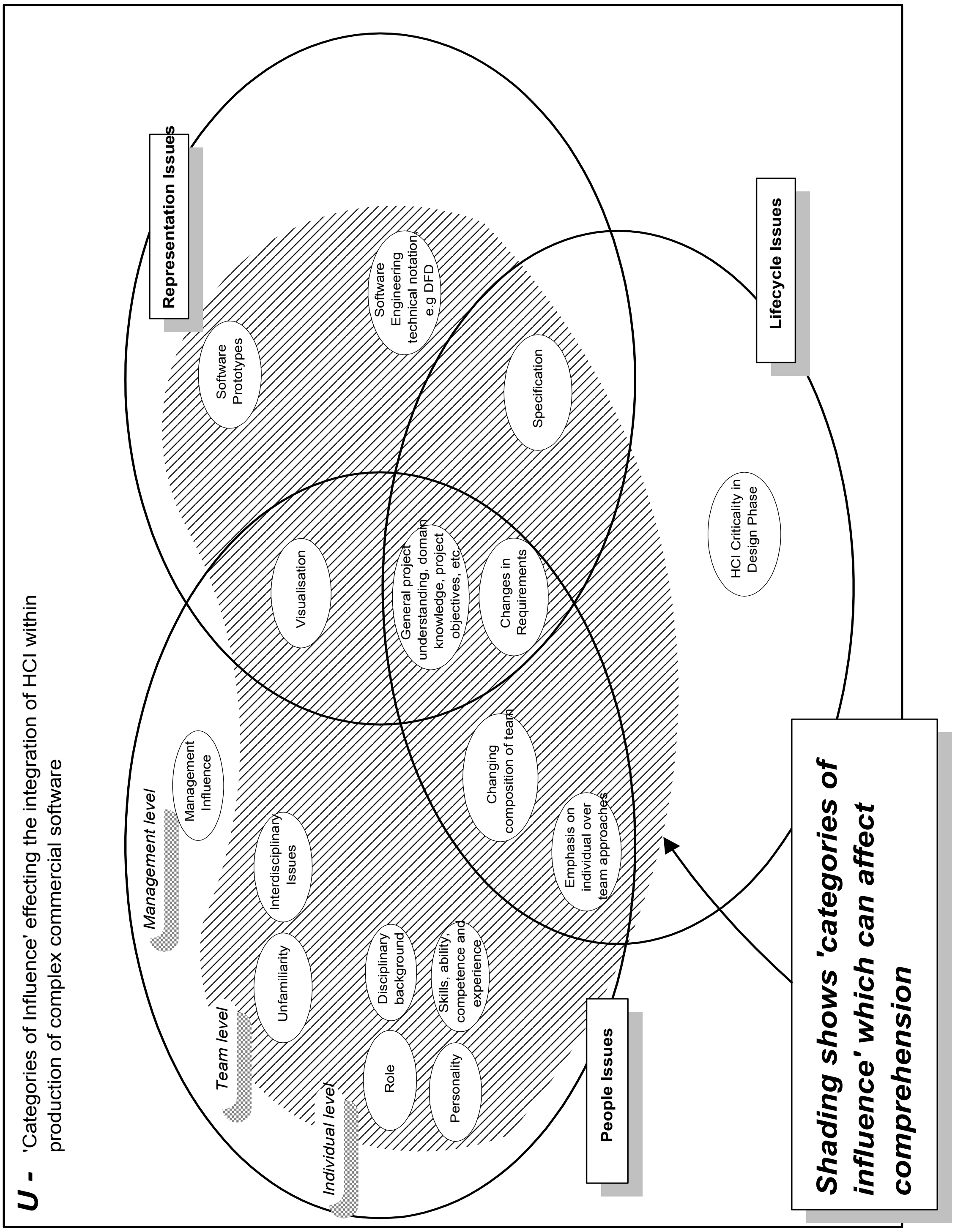

U - 'Categories of Influence' effecting the integration of HCI within production of complex commercial software
Representation Issues
Lifecycle Issues
People Issues
Software Prototypes
Software Engineering technical notation e.g DFD
Specification
HCI Criticality in Design Phase
Visualisation
General project understanding, domain knowledge, project objectives, etc.
Changes in Requirements
Changing composition of team
Emphasis on individual over team approaches
Management level
Management Influence
Interdisciplinary Issues
Team level
Unfamiliarity
Disciplinary background
Skills, ability, competence and experience
Individual level
Role
Personality
Shading shows 'categories of influence' which can affect comprehension

Figure 2.7 ‘Categories of Influence’ within the Conceptual Model that can effect comprehension

2.9.2.7 'Categories of Influence' That Are Features of the Commercial Arena

Some of the 'categories of influence' within the Venn Diagram representation provide a context for the comprehension difficulties that exist and are apparently features of the complex commercial software development arena. Those features considered dominant are outlined below.

2.9.2.7.1 Individuals' Characteristics Appeared to Effect Their Ability to Understand, Visualise and Explain Software

Categories of influence, including 'Personality', 'Skills, Competence and Experience', and 'Disciplinary Background', describe the characteristics of an individual in the development team. A combination of these characteristics appeared to effect a person's ability to understand and explain various software concepts. A particularly important characteristic seemed to be an individual's ability to visualise (see sections 2.9.1.6.1 and 2.9.2.4.1 and the 'Visualisation' category in section 2.9.1.6.4)

Curtis *et al.* (1987) and Brooks (1986) identified the existence of individuals with particularly strong conceptualisation ability (see section 1.10.2.1 on individual differences). This implies that this ability is variable, and findings from the ethnographic study confirm this.

2.9.2.7.2 The Software Team Was Mixed Ability

The 'Changing Composition of Team' category highlights the fact that commercial practice dictates resource availability and the deployment of individuals on software teams. For example, on both projects studied there was a change in team members following the production of the specification. This change in personnel was based on resource availability, rather than team building, i.e. a team member's personal characteristics or their familiarity with other people in the existing team was not an issue. It was therefore apparent that the software team exhibited a mixture of abilities (a number of data samples throughout sections 2.9.1.6.1 and 2.9.2.4.1 illustrate this).

Mixed ability teams are not described in the literature. However, as discussed in section 1.10.2.1, the existence of individual differences found among commercial programmers implies that software teams are mixed ability. This ethnographic study supports this supposition.

2.9.2.7.3 Discontinuity of the Team Created a Comprehension Burden

Discontinuity made it hard for the team to maintain a consistent mental model of the software and minimise misunderstandings. The composition of the team needed to change in order to optimise the commercial allocation of resources (as described in sections 2.9.1.6.7 and 2.9.2.4.7). On both projects, a number of programmers joined the team when the specification had been written (several months after the start of the projects). The communication burden on the specification document (see the 'Specification' category in section 2.9.1.6.6) illustrates the comprehension burden that exists when the team composition changes.

Problems caused by changing team composition are described in section 1.10.3. Grudin (1991) pointed out the extreme case where design and development are carried out by different teams. "Design drift" caused by new people joining a team, was described by Harker (1991). Grudin (1996) concurred, and highlighted the burden this causes the rest of the team as they have to explain aspects of the software to new team members. The findings of this ethnographic study confirm these reports.

2.9.2.7.4 Members of Interdisciplinary Teams Experienced Difficulties Understanding Each Other

Members of the interdisciplinary teams studied experienced difficulties in understanding each other. This was apparently by virtue of their disciplinary background (e.g. see 'Disciplinary Background' category within section 2.9.1.6.1) and seemed to be contributed to by their role priorities (e.g. see 'Interdisciplinary Issues' category in section 2.9.1.6.1). For example, the 'Disciplinary Background' category within section 2.9.2.4.1 provides an example of a fundamental problem with programmers understanding the importance of user tasks during a design discussion about modal dialogs.

Kim (1990) and Erickson (1990) have reported that disciplines each have different perspectives and priorities and that these can conflict (see section 1.10.3). Kim emphasised that people from diverse disciplines can have fundamental problems understanding each other. Ethnographic findings confirm that such problems occur.

2.9.2.7.5 Requirements Changes Introduced Scope for Misunderstanding

That changes in requirements appear to be an inevitable feature of software development was a clear finding from both projects and seemed to be fuelled by the processes enforced by the lifecycle and commercial considerations. For example, the lifecycle demanded that effort estimates are made at the start of a project in the face of considerable unknowns. As these unknowns become known, the requirements have to change but the estimates often cannot, causing tension within the team. (see the 'Changes in Requirements' category of section 2.9.1.6.5).

When requirements change, a primary issue is how the team are kept informed of these changes (e.g. see 'Changes in Requirements' category in section 2.9.2.4.5). Both projects had several versions of specification documents and consequently team members were buried under paper. Failure to effectively communicate the changes was observed to cause misunderstandings. For example, changing requirements were a significant feature of project 1 resulting in team members having a poor shared understanding of the system being developed, thus causing many misunderstandings (see the 'Changes in Requirements' category of section 2.9.1.6.5).

Many researchers believe that changes in requirements are inherent in software projects (see section 1.4.2). However, the scope for misunderstanding within the team which such changes create, which was found in the ethnographic investigation, is not reported in the literature.

2.9.2.7.6 Specification Documents Were an Inevitable Compromise Due to their Diverse Uses and Readerships, but Were Considered Essential Milestones

Specification documents were found to have diverse uses and readerships in the development process for both projects studied. The diversity of uses and readers of the specification appeared to make the document a compromise in all its intended purposes. However, the specification was also regarded as an essential milestone in the software lifecycle. (see the 'Specification' category of section 2.9.1.6.6).

The continual dominance of the waterfall lifecycle and fixed price/time software contracts reinforce the importance of the software specification in current commercial software development (see section 1.6). It is also apparent from literature that many people believe written specifications to be open to misinterpretation and are an ineffective means of supporting collaboration within software teams (see section 1.11.3.2). This ethnographic investigation confirmed the apparently essential need for written specifications in commercial software practice but found that the specification is a poor working document for the team, because of its compromised effectiveness in all of its modes of use.

2.9.2.8 'Categories of Influence' Which Could be Targeted for Improving Comprehension

Other 'categories of influence' that lie within the comprehension sphere are intended to facilitate understanding. Such areas could be realistically targeted for improvement of comprehension within the team (figure 2.8), within commercial constraints, and subject to the features of commercial projects outlined in section 2.9.2.7. The following 'categories of influence' are therefore aspects of a software project designed to facilitate comprehension, but which have been identified as potential targets for improvement.

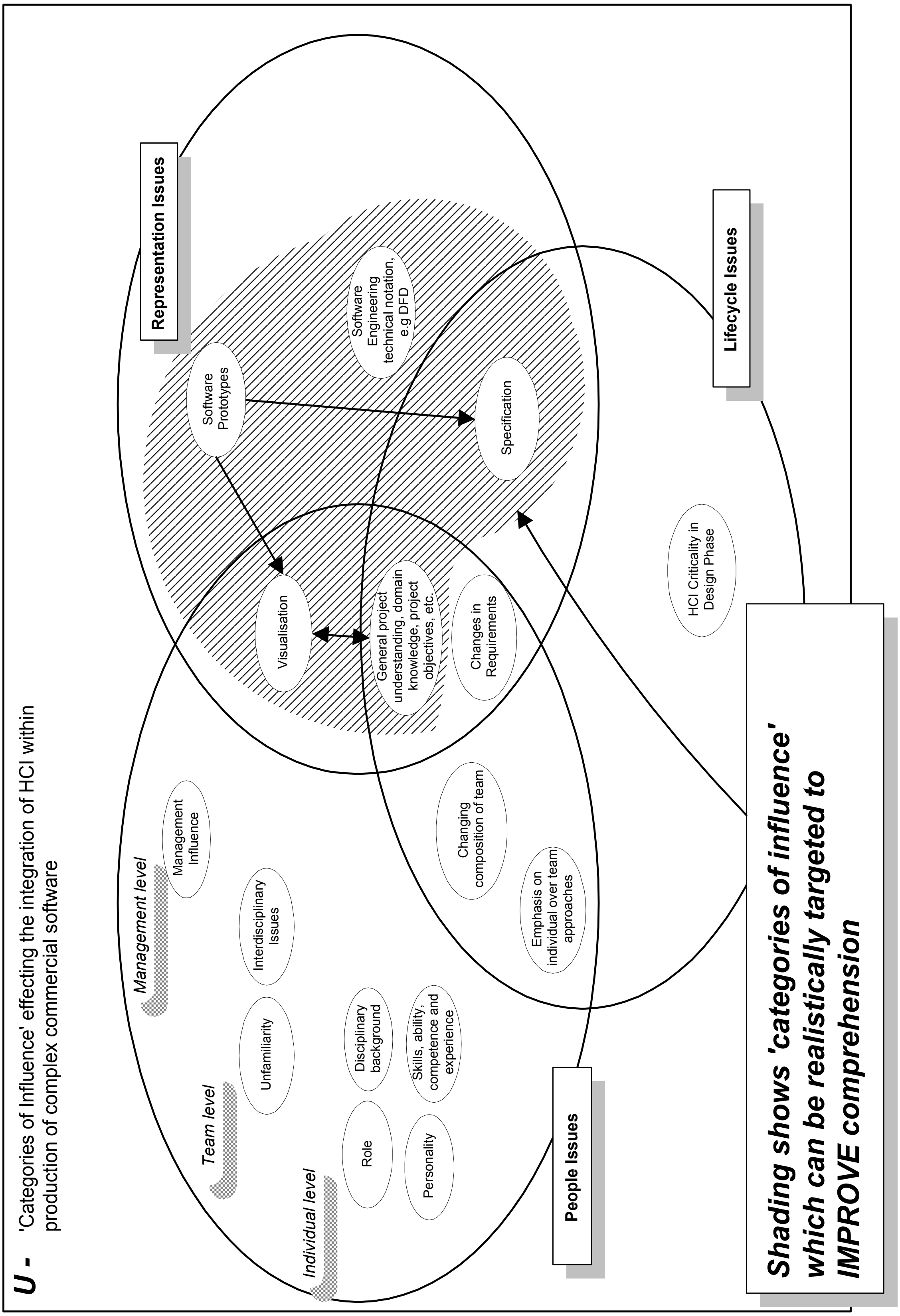

U -
'Categories of Influence' effecting the integration of HCI within production of complex commercial software
Representation Issues
Lifecycle Issues
People Issues
Management level
Team level
Individual level
Software Prototypes
Software Engineering technical notation, e.g DFD
Specification
Visualisation
General project understanding, domain knowledge, project objectives, etc.
Changes in Requirements
HCI Criticality in Design Phase
Management Influence
Interdisciplinary Issues
Unfamiliarity
Disciplinary background
Skills, ability, competence and experience
Role
Personality
Changing composition of team
Emphasis on individual over team approaches
Shading shows 'categories of influence' which can be realistically targeted to IMPROVE comprehension

Figure 2.8 ‘Categories of Influence’ which could be realistically targeted to improve comprehension

2.9.2.8.1 Software Engineering Technical Representations

Software engineering technical representations like DFDs required training to interpret and failed to accept some of the realities of complex commercial software development illustrated by the features outlined in section 2.9.2.7. In particular, issues arising from the interdisciplinary (section 2.9.2.7.4) and mixed ability (2.9.2.7.2) nature of the team and apparent variability in individuals' visualisation abilities (2.9.2.7.1) are not addressed by these traditional representations.

Representation and notation is briefly discussed in section 1.11.3.1. The need to improve existing representations (software engineering technical representations are the predominant existing representations) is well recognised. Curtis (1988) and Brooks (1995) believe that multiple forms of representation are required within software development. The ethnographic investigation supports this view as not all representations used were suitable for all team members.

It is believed that improvement in this area would need to be in the form of a more accessible and understandable representation, which would be appropriate for people from diverse disciplines and different ability levels. The likelihood of developing such a generally understandable representation following the format of traditional software engineering technical representations seems remote.

2.9.2.8.2 Software Specifications

As well as being a feature of commercial software development (described in section 2.9.2.7.6), the specification formed a working document for the software team. As described in section 2.9.2.7.6, the multiple uses and diverse readers of the specification caused it to be flawed, particularly in its use as a working document for the team.

It is believed that improvement to the quality of the authoring of the specification could feasibly generate some improvements to its utility. Unfortunately, such advanced authoring skills are apparently rare inside a software team.

2.9.2.8.3 General Project Understanding

General project understanding, for example of the knowledge of the application domain or the project objectives, was rarely explicitly stated or available to all team members (see 'General Project Understanding…' in section 2.9.2.4.5 where team members' reliance on second-hand information is discussed). Furthermore, some team members seemed to think that such information was of no use to them (also covered in the section previously referred to).

The need for team members to share an understanding of the software they are jointly producing has partly been established by the literature pertaining to conceptual integrity (see section 1.11.2). However, this literature is focused on team members sharing a conceptualisation of the software being produced and does not specifically consider the need for them to share a general understanding of the project application domain or objectives (perhaps this is partially implicit). The literature available on the subject of design rationale (see section 1.11.3.4) also demonstrated research interest in providing team members with underlying information about the designs they are implementing. Like the literature concerning conceptual integrity, the design rationale literature does not specifically cite a need for team members to share a general project understanding. Therefore, a specific need for all team members to share a general understanding of the wider issues surrounding the software production identified in ethnographic studies was not evident from the literature. The finding that some team members believed that this kind of information was of no use to them was also not apparent from the literature.

It is believed that improvements to the distribution of such understanding within the team could improve team members' day-to-day decisions and help them to better understand the software being produced.

2.9.2.8.4 Representations to Facilitate Software Visualisation Within the Team

Although members of the mixed ability team exhibited differing levels of visualisation ability, some representations utilised seemed to make the most of these abilities (e.g. scenarios and prototypes - see the 'Software Prototypes' category in section 2.9.1.6.2).

Although not often described in the literature applied to software teams, some representations have been found to provide a common currency, which is understandable by both developers and users. Although this is different to collaboration within a software team, developers and users often have diverse skills and disparate computing abilities, so could be considered a model of an extremely mixed ability team. In section 1.11.3.1, stories (Erickson, 1996) and scenarios (Carey *et al.*, 1991; Carroll and Rosson, 1990; Karat and Bennett, 1991; Nielsen, 1993) are described as representations suitable for communication between developers and users.

During the project 2 development, scenarios were used as a means of presenting prototype walkthroughs. They were found to complement prototyping as an effective means of helping all members of the team (and others) to visualise the proposed software. Although this specific use of scenarios is not evident from literature, the results found in the case studies are unsurprising because the literature suggests that scenarios facilitate communication between developers and users.

It is believed that providing better means for team members' to visualise the proposed software is likely to help them maintain a better mutual understanding of the software (in other words, better maintain conceptual integrity).

2.9.2.8.5 Prototypes and Design Rationale

During both software projects, the use of visual prototypes provided a common representational currency which helped team members from a variety of disciplines, with a mixture of abilities, visualise and understand the software. Such prototypes were also used to illustrate the specification document to help team members and others to understand and critique it. However, using prototypes to facilitate comprehension within the team was also found to be a flawed approach. Prototypes were found to be inaccessible (see the 'Software Prototypes' category in section 2.9.1.6.2) and ambiguous - because programmers would often focus on an incidental aspect of the prototype (like dummy data) rather than a complex interaction (see the data sample under the 'Interdisciplinary Issues' category of section 2.9.1.6.1).

In section 1.11.3.3.2, the use of prototyping for visualisation within the team was suggested (Miller-Jacobs, 1991; Wagner, 1990) and claims were made that prototyping could be a common reference for members of the software team. Section 1.11.3.3.3 describes the limited attention that using prototypes to facilitate comprehension within software teams has received (e.g. Gladden, 1982; Heckel, 1991; Miller-Jacobs, 1991; Rudd and Isensee). Many of these claims in the literature are not described in any depth and are largely unsubstantiated.

In section 1.11.3.2 the use of prototypes to support and explain what is written in specification documents is discussed, and some researchers suggest that prototypes have this potential (e.g. Gomaa, 1983; Heckel, 1991; Wilson and Rosenberg, 1988).

Findings from the ethnography support suggestions in the literature that prototyping can facilitate visualisation and can provide a common reference to members of a software team. Further findings confirm that prototypes can be an effective means of helping people to understand what is contained in a written specification.

An important ethnographic finding is that prototypes may have been inaccessible to team members. Prototypes were constructed to be demonstrated by the prototype developer (the HCI designer), because the operation of the façade they presented often utilised discreet key presses and hidden locations on screen (see the 'Software Prototypes' category in section 2.9.1.6.2). Preece *et al.* (1994) describe such prototypes as chauffeur-driven (see section 1.11.3.3.1). This inaccessibility is not recognised in the literature.

An equally important finding from the ethnographic studies is that prototypes were sometimes ambiguous and open to interpretation (and misinterpretation) by team members. Although prototyping is superficially suggested (sometimes even naively suggested, e.g. the "picture paints a thousand words" arguments of Gladden (1982), and Rudd and Isensee (1994)) as a means of facilitating comprehension within the team, the literature does not acknowledge the potential ambiguity of the representation. Ethnographic findings show that prototypes can be ambiguous, and this is not recognised in the literature.

Using prototypes to facilitate comprehension and as a communication medium was further found to be lacking because it failed to facilitate an explanation of the rationale underlying the HCI design (see the ‘Software Prototypes’ category in section 2.9.1.6.2). The rationale was usually stored in the head of the HCI designer only, evident from the fact that team members would nearly always ask questions of the HCI designer, rather than merely refer to the prototype. Design rationale was rarely noted down or explained to other team members in advance of queries arising. Lack of an explanation of design rationale meant that team members were not in a position to understand the reasoning behind various elements of the user interface and therefore had no basis from which to negotiate or offer design alternatives (which may have been easier to implement or provided a better solution) – (see the ‘Software Prototypes’ category in section 2.9.1.6.2).

Section 1.11.3.4 describes the use of design rationale within software teams. Ethnographic findings suggest that design rationale was usually stored in the head of the HCI designer. Similar claims were made by Conklin and Burgess-Yakemovic (1996). The absence of design rationale in the prototype itself confirms Moran and Carroll’s suggestion (1996) that artifacts produced by the design process do not inherently indicate the reasoning underlying their design.

It is believed that prototypes have a great potential for helping a diversity of team members and other stakeholders to visual the software under development in a way that they can all understand. However, there are a number of flaws in this use of prototypes, that have not been addressed in the literature, but which have the potential to cause severe difficulties in the software development process.

2.9.2.9 Summary and Final Commentary of Focussed Observations

The focused observations began with a write-up of the ethnographic study of project 2. The researcher then carried out a basic comparison between project 1 and project 2. The main result apparent from this comparison was that project 2 progressed much more smoothly than project 1. The project 2 team apparently maintained a more consistent understanding of the software (conceptual integrity) and major misunderstandings (which had been a feature of project 1, often requiring re-work) were reduced. From this analysis, ethnographic experience and the analysis afforded by the data collection and write-up, the researcher focused down to the key theme of ‘comprehension’ as being related to the majority of the ‘categories of influence’.

The ‘categories of influence’ that relate to comprehension were then identified on the Venn Diagram. These categories were split into apparently immutable features of commercial software development that have an effect on comprehension, and aspects of the setting that intended to facilitate comprehension. The latter were proposed as targets for improving the comprehension difficulties that appeared to be prevalent in the projects observed.

Focusing further, it is apparent that the targets for improvement to comprehension related to the interaction between the HCI designer and the programmers, and how they maintained a shared understanding of the design. This focus illustrated the potential comprehension difficulties that can exist between team members in these roles.

The continuing focus described above gradually achieved through the focused observations stage is illustrated by the funnel diagram in section 2.8.6.2.

For clarity, the following sections summarise which findings from the investigation correspond to existing research results described in the literature, and which are solely findings from this investigation so far.

2.9.2.9.1 Findings that Confirm Existing Research

Findings from the ethnographic investigation which confirmed claims and reported findings from the literature are summarised here and described in detail in sections 2.9.2.7 and 2.9.2.8.

- Individuals' characteristics appeared to effect their ability to understand, visualise and explain software.

- Software teams appear to be comprised of individuals with differing levels of ability (it is argued by the literature review in this thesis that the existence of individual differences among commercial computer programmers commonly leads to mixed ability teams).

- Discontinuity of the team appeared to create a comprehension burden.

- Members of interdisciplinary teams apparently experience difficulties understanding each other.

- Specification documents were found to be a fundamental milestone in the software development process.

- The diversity of uses and readers of the specification appeared to make the document a compromise in all its intended purposes, including as a working document for the software team. It therefore failed to help readers visualise or understand the proposed software.

- Prototyping appears to be an effective means to facilitate visualisation, comprehension and provide a common reference to members of a software team.

- Prototypes can be an effective means of helping people to understand what is contained in a written specification.

- Design rationale was usually stored mainly in the head of the HCI designer and was not conveyed by artifacts from the design process, like the prototype.

2.9.2.9.2 Original Findings from Ethnographic Investigation

Findings from the ethnographic investigation, which are believed to be original are summarised here and described in detail in sections 2.9.2.7 and 2.9.2.8.

- Requirements changes throughout the software development process introduced further scope for misunderstanding within the software team.
- There is a need for all team members to share a general understanding of the wider issues surrounding the software project, (e.g. an understanding of the application domain), as it is apparent that technical decisions are often made by team members without such knowledge.
- Some team members believed that a general understanding about the software project was of no use to them.
- Scenarios were found to complement prototyping as an effective means of helping all members of the team (and others) to visualise the proposed software.
- Prototypes were inaccessible to team members because of their design and development for chauffeur-driven operation.
- Prototypes were sometimes ambiguous and open to interpretation (and misinterpretation) by team members.

The focus on comprehension, both the immutable features of a software setting and the identification of targets for improvement of comprehension within the team, are also original findings. As is the observation that the targets for improvement centre around the interaction between the HCI designer and programmers and the conceptual integrity of the design they share a vision of. Therefore, the selective observations described in the following section show a re-directed focus of this study to the interaction between the HCI designer and the programmers.

2.9.3 Selective Observations of the Interaction Between the HCI Designer and the Programmers

This section covers the ‘selective observations’ of the ethnographic study (see section 2.8.6.3 for a further description of this aspect of the methodology). At the bottom of the focused observations section of the funnel, the focus of the ethnography has centred on the interaction between the HCI designer and the programmers. Specifically of interest are the various means used by the HCI designer and programmer to share an understanding of the design.

Some specific inferences made during selective observations are supported by findings from the more broadly focused investigation of the setting, higher up the funnel (i.e. at the descriptive or focused observations stage). However, at this selective observations stage in the funnel, the interaction between HCI designer and programmers is put under the microscope. The link between the focused and selective observations stages of the funnel is that findings from focussed observations provided the microscope’s focus for the selective observations. This means that the description of the interaction between HCI designer and programmers comes from the directed re-focusing of the funnel, rather than from a specific distillation of findings thus far.

Whilst still in the field working on project 2, the researcher began analysing the interaction between the HCI designer and the programmers; this analysis forms the basis of the selective observations. Typical HCI techniques (learned whilst performing the role of HCI designer during the participant-observation) were used to analyse and describe this interaction. Initially the roles, responsibilities and objectives of the HCI designer and programmer roles are briefly described. Next, a fundamental conflict between these roles is identified. The following section then analyses the communication between the HCI designer and programmers, across all phases of the project. This is then further focused, firstly onto the nature of communication during the implementation phase, and secondly, onto the volume of communication at this phase. Finally, at the base of the funnel, specific problems in the interaction between the HCI designer and the programmers are described.

2.9.3.1 Roles, Responsibilities and Objectives

This section describes the essence of the role of the HCI designer and the programmer, and their main responsibilities. The broad objectives of each role are then outlined.

2.9.3.1.1 HCI Designer

The HCI designer is responsible for fighting the end-user’s corner throughout the software design and implementation process. The main goal is to design the user interface and they are therefore responsible for the resulting human-computer interaction with the users.

Achieving ease of use and functionality appropriate to the user's tasks and the client requirements in the software produced are priorities for the HCI designer.

In addition to their user interface design responsibility the HCI designer must also ensure that HCI design intent is followed during implementation in order to realise their software vision.

The broad objectives of the HCI designer's role are:

1. **Gain an understanding of users, their tasks, organisation and domain.** (including software requirements)

2. **Design an appropriate user interface to the software.**

3. **Ensure that the implementation follows HCI design intent.**

2.9.3.1.2 Programmer

In this context, the term 'programmer' also includes the IT designers. They are often senior programmers responsible for the High Level Design (HLD) of the software and have a hands-on programming role as well. Programmers are responsible for implementing the software that has been specified, including the user interface, within tight timescales.

The programmers are responsible for designing the internal structure of the proposed software and then implementing this in a way which complies with HCI design intent and allows the user interface to function as designed.

Programmers' priorities are to produce code within their own time estimates (e.g see data samples in the 'Changes in requirements' category of section 2.9.1.6.5). Other priorities include producing code to an agreed coding standard, achieving a high standard of documentation and writing code in an easily maintainable way. Technical elegance of the underlying program is often more important to the programmer than visual or interactive considerations.

Because programmers are required to deliver code to tight timescales, it is their responsibility to ensure that the user interface design is viable. In cases where this is found not to be the case, programmers need to suggest alternatives and negotiate with the HCI designer.

The broad objectives of the programmers' role are:

1. **Gain an understanding of the user interface design, software requirements, project goals and the domain.**

2. **Software design (internal), high level design (HLD) and low level design (LLD**).

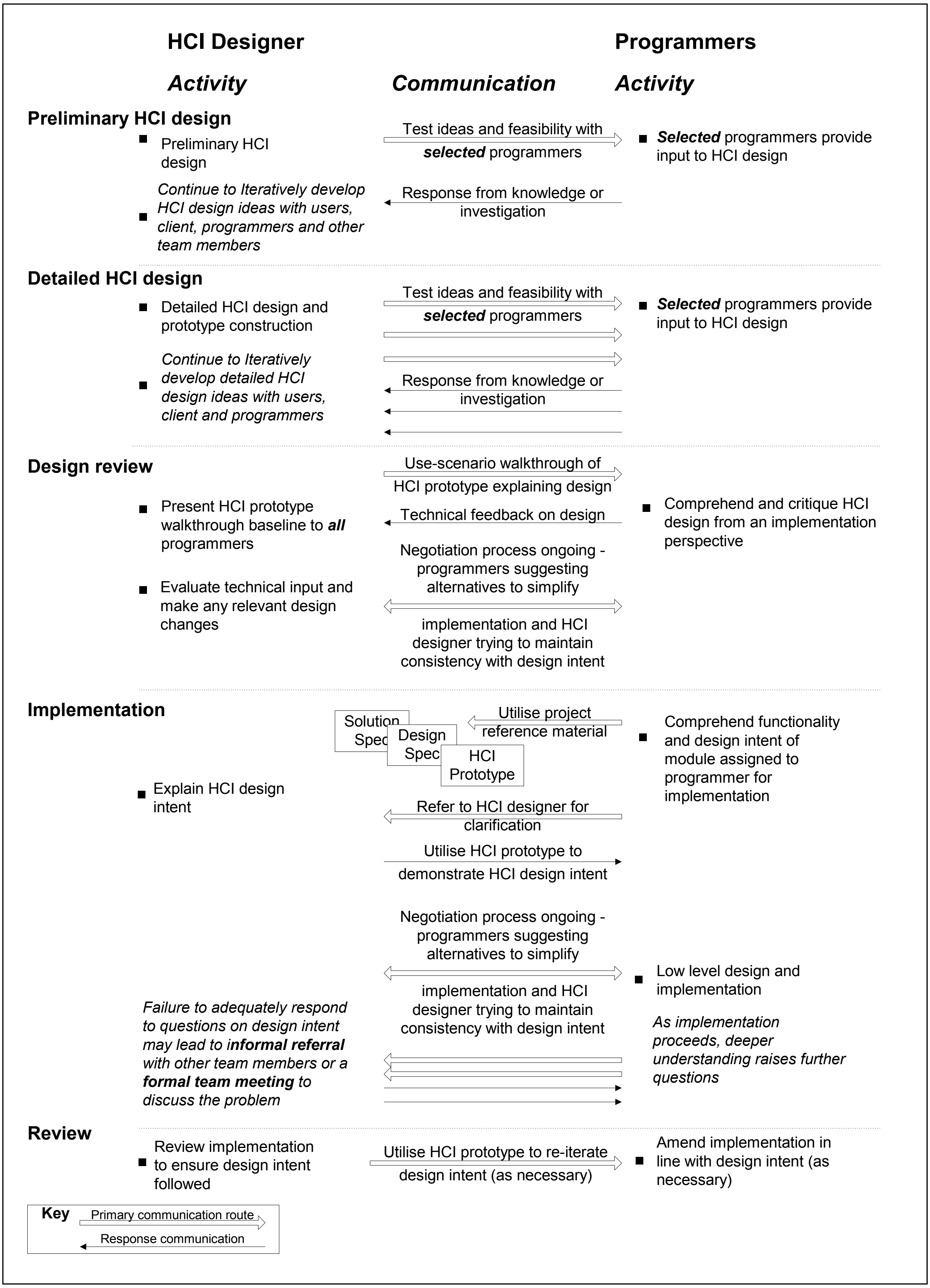
3. **Implementation of code.**

2.9.3.2 Fundamental Conflict Between the HCI Designer and Programmer Roles

The priorities of the programmer and the HCI designer are often in conflict (e.g. see data sample in the 'Role' category of section 2.9.1.6.1). This is primarily because a user interface, which is optimal for a user, is often quite hard to implement, especially where Graphical User Interfaces (GUIs) are involved. Compromise between usability and technical feasibility is a regular issue in HCI designer and programmer collaboration. Such compromise often becomes difficult when internal software structures (the responsibility of the programmer) are rejected because of undesirable effects they have on the user interface (the responsibility of the HCI designer). Similar debates occur when the user interface has to suffer because of internal software structures.

2.9.3.3 Communication between HCI Designer and Programmer Across Project Phases

Figure 2.9 provides an outline of the type of communication that occurs between the HCI designer and the programmers (including IT designers) across the project phases when they need to collaborate.



HCI Designer — Activity | Communication | Programmers — Activity

Preliminary HCI design

- Preliminary HCI design
- *Continue to Iteratively develop HCI design ideas with users, client, programmers and other team members*

Test ideas and feasibility with ***selected*** programmers

Response from knowledge or investigation

- ***Selected*** programmers provide input to HCI design

Detailed HCI design

- Detailed HCI design and prototype construction
- *Continue to Iteratively develop detailed HCI design ideas with users, client and programmers*

Test ideas and feasibility with ***selected*** programmers

Response from knowledge or investigation

- ***Selected*** programmers provide input to HCI design

Design review

- Present HCI prototype walkthrough baseline to ***all*** programmers
- Evaluate technical input and make any relevant design changes

Use-scenario walkthrough of HCI prototype explaining design

Technical feedback on design

Negotiation process ongoing - programmers suggesting alternatives to simplify implementation and HCI designer trying to maintain consistency with design intent

- Comprehend and critique HCI design from an implementation perspective

Implementation

- Explain HCI design intent

*Failure to adequately respond to questions on design intent may lead to **informal referral** with other team members or a **formal team meeting** to discuss the problem*

Solution Spec

Design Spec

HCI Prototype

Utilise project reference material

Refer to HCI designer for clarification

Utilise HCI prototype to demonstrate HCI design intent

Negotiation process ongoing - programmers suggesting alternatives to simplify implementation and HCI designer trying to maintain consistency with design intent

- Comprehend functionality and design intent of module assigned to programmer for implementation
- Low level design and implementation

As implementation proceeds, deeper understanding raises further questions

Review

- Review implementation to ensure design intent followed

Utilise HCI prototype to re-iterate design intent (as necessary)

- Amend implementation in line with design intent (as necessary)

Key

Primary communication route

Response communication

Figure 2.9 Outline of the type of communication that occurs between HCI designer and programmers across project phases

During the HCI design phase, the HCI designer needs to come up with a user interface design that is feasible to implement. As well as using their own technical judgements, the HCI designer uses selected programmers to test ideas out and to suggest alternatives. Only selected programmers are used in this way, as some appear to have a better aptitude for this type of creative work than others. This reflects the individual differences evident among programmers, leading them to play different informal roles within the team (see section 1.10.2.1).

During the design review phase, the HCI designer presents the external design of the software (the user interface) to all of the programmers that will be involved in the implementation. The HCI designer uses the baselined prototype and scenarios of use to explain the HCI design intent of the proposed software (e.g. see 'Software Prototypes' in sections 2.9.1.6.2 and 2.9.2.4.2). During the review, it is the programmers' job to comprehend the HCI design intent and gain a partial understanding of the underlying users' tasks (although the latter is not essential to their job). At the design review, programmers will begin to assess the most obvious implementation implications that the external software design (i.e. HCI design) has on the internal software design. However, as is indicated in literature, many of these implications will not become apparent until the implementation is under way, causing external design issues to be reconsidered throughout the implementation phase (see section 1.4.5). Some of the implications identified at this stage will bring about a negotiation process between HCI designer and the programmers (e.g. see 'General Project Understanding' in section 2.9.2.4.5). Programmers may be concerned about certain difficult aspects of the user interface and suggest alternatives. The HCI designer assesses the impact of the alternatives on the overall HCI design intent and may present a compromise external design.

The communication between the HCI designer and programmers is at its most intense during the implementation phase. During this phase, the programmers' interest in aspects of the proposed functionality is at its peak, because they now have to begin construction. When a programmer has been assigned an element of the software to implement, they must first comprehend the proposed functionality. The first step in gaining this understanding is to utilise project reference material such as the solution specification (or requirements specification), the design specification and the HCI prototype. If clarification is required, the programmer will refer to the HCI designer, who will often utilise the HCI prototype to demonstrate the HCI design intent of the proposed functionality (e.g. see data samples in 'Software Prototypes' category in section 2.9.2.4.2). As in the previous project phase, if the external design for an element of the implementation proves too difficult to realise, programmers may suggest alternatives for the HCI designer to consider, and a negotiation process begins.

In the review phase, the HCI designer assesses implemented aspects of the final software against their conceptual model (in Norman's (1986) terms, the 'Design Model', see section 1.11.2). Where the implementation has deviated from the HCI design intent, the HCI designer may utilise the prototype to explain to the programmers how the functionality should have been implemented. Changes to the implemented code that are required, for this reason and others, are usually prioritised. The prioritisation itself is also the subject of some negotiation.

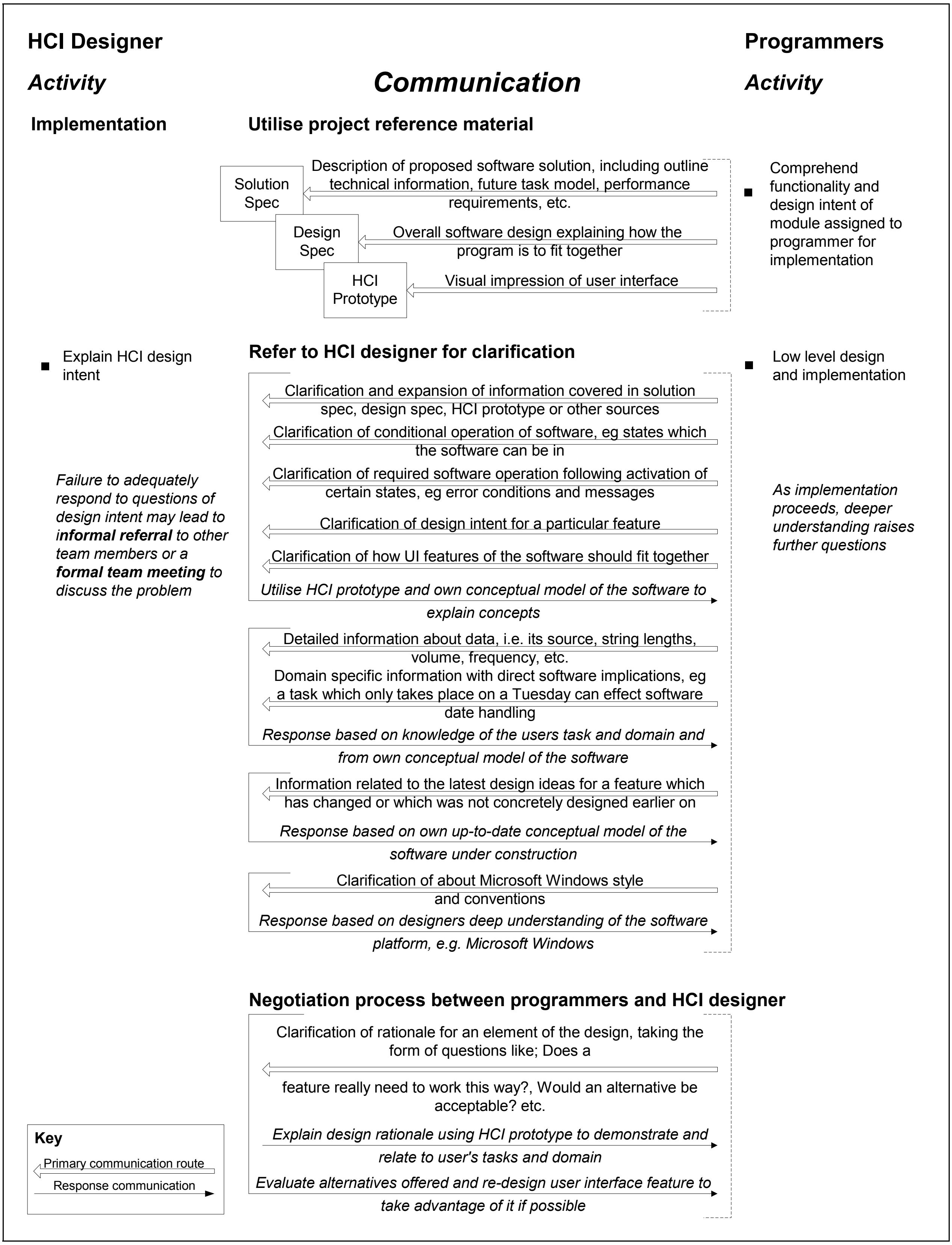
2.9.3.4 Detailed Analysis of Communication between HCI Designer and Programmer During Implementation Phase

As mentioned in the previous section, during the implementation phase programmers must gain a detailed understanding of the proposed software as they have now been assigned aspects of it to implement. This section analyses the communication in the implementation phase in more detail and is supported by the diagram in figure 2.10.

By the implementation phase, it is usual for some project reference material to have been produced. In fact, the completion of the solution specification (or requirements specification) usually signifies the start of the implementation phase. The HCI prototype is also usually completed by this time, but the design specification (the overall design of the software internals) may only just have been started, or it may be completed. This project reference material is the likely starting point for the programmer to gain a detailed understanding of the software aspect to be implemented. The solution specification provides descriptions of the proposed software solution, including outline technical information, future task model (a model of the proposed structure of the user's tasks when the software has been installed), software performance requirements, etc. The design specification provides information on the overall design of the software internals and should clarify how various aspects of the software will fit together, enabling the programmer to assess where the aspects they are to implement fit into the overall picture. The programmers may also make use of the HCI prototype to gain a visual impression of the user interface and the proposed software as a whole (however, the prototypes can prove somewhat inaccessible to programmers - see the 'Software Prototypes' category in section 2.9.2.4.2).

During the implementation phase of project 2, it became clear that the project reference material was insufficient to convey a detailed enough understanding of the implementation to the programmers (see section 1.11.3.2). Performing a role similar to that described by Curtis (see section 1.11.2 and 1.10.2.1) as 'Super-Conceptualiser' or 'keeper of the project vision', the HCI designer was referred to for clarification of a multitude of issues (e.g. see data sample in the 'Emphasis on Individual over Team Approaches' category in section 2.9.2.4.7), such as:

- clarification and expansion of project reference material;
- clarification of conditional software operation;
- clarification of required software response to a series of states, e.g. error conditions and messages;
- clarification of HCI design intent for a feature;
- clarification of how user interface features should fit together.



HCI Designer

Activity

Communication

Programmers

Activity

Implementation

Utilise project reference material

Solution Spec

Description of proposed software solution, including outline technical information, future task model, performance requirements, etc.

Design Spec

Overall software design explaining how the program is to fit together

HCI Prototype

Visual impression of user interface

- Comprehend functionality and design intent of module assigned to programmer for implementation

- Explain HCI design intent

Refer to HCI designer for clarification

- Low level design and implementation

Clarification and expansion of information covered in solution spec, design spec, HCI prototype or other sources

Clarification of conditional operation of software, eg states which the software can be in

Clarification of required software operation following activation of certain states, eg error conditions and messages

Clarification of design intent for a particular feature

Clarification of how UI features of the software should fit together

Utilise HCI prototype and own conceptual model of the software to explain concepts

*Failure to adequately respond to questions of design intent may lead to **informal referral** to other team members or a **formal team meeting** to discuss the problem*

As implementation proceeds, deeper understanding raises further questions

Detailed information about data, i.e. its source, string lengths, volume, frequency, etc.

Domain specific information with direct software implications, eg a task which only takes place on a Tuesday can effect software date handling

Response based on knowledge of the users task and domain and from own conceptual model of the software

Information related to the latest design ideas for a feature which has changed or which was not concretely designed earlier on

Response based on own up-to-date conceptual model of the software under construction

Clarification of about Microsoft Windows style and conventions

Response based on designers deep understanding of the software platform, e.g. Microsoft Windows

Negotiation process between programmers and HCI designer

Clarification of rationale for an element of the design, taking the form of questions like; Does a feature really need to work this way?, Would an alternative be acceptable? etc.

Explain design rationale using HCI prototype to demonstrate and relate to user's tasks and domain

Evaluate alternatives offered and re-design user interface feature to take advantage of it if possible

Key

Primary communication route

Response communication

Figure 2.10 HCI designer/programmer communication during the implementation phase

The HCI designer utilises the HCI prototype as a concrete embodiment of their conceptual model to provide the necessary clarification to the programmer (e.g. see 'Visualisation' category in section 2.9.2.4.4). Alternatively, the HCI prototype may be used to work through and visualise the issues that the programmer has raised (e.g. see 'Visualisation' category in section 2.9.2.4.4). Further clarification required may relate to data items, for example:

- clarification of the source, type, length, frequency and range of data items;
- clarification of domain events which affect the data, e.g. a task which only takes place on a Tuesday can affect software date handling and range-checking.

The HCI designer responds to such questions based on knowledge of the users' tasks and domain combined with their own conceptual model of the software. Clarification is also likely to be required on elements of the software which have recently been changed. As the majority of changes involve HCI redesign work, the HCI designer is best placed to explain the changes based on their own up-to-date conceptual model of the software, possibly making use of the prototype to facilitate explanation. Another common form of clarification occurs when implementation detail becomes greater than the level of detail represented in the prototype. At this stage, detailed and subtle aspects of platform style guides (e.g. Microsoft Windows style) and house style guides may require interpretation. If style conventions are being broken, the rationale for this is also a subject of some debate. The HCI designer responds to such questions from programmers based on a deep knowledge of the software platform style and house style.

As the implementation proceeds, it may become apparent to programmers that aspects of the user interface may be hard to implement in the way that the HCI designer intends. This usually starts a negotiation process between the HCI designer and programmer. The programmer may start such a negotiation with a question like "does this really have to work in this way?" or, "are you sure we need to do this?" Ultimately a compromise, which is easier to implement and yet fits with HCI design intent, is usually found.

2.9.3.5 An Illustration of the Volume of Communication between HCI Designer and Programmer During Implementation Phase

Figure 2.11 makes three key points about communication and comprehension activities within the implementation stage. The first point is that the programmers' utility of the HCI prototype to support their gaining an understanding of the software under production is low (e.g. see 'Software Prototypes' category of section 2.9.2.4.2). Secondly, there is a high volume of communication between the programmers and the HCI designer both for clarifications and during negotiation processes (e.g. see 'Software Prototypes' category of section 2.9.2.4.2). Thirdly, a high proportion of implementation alternatives suggested by programmers during the negotiation process were uninformed and therefore receive negative responses from the HCI designer.

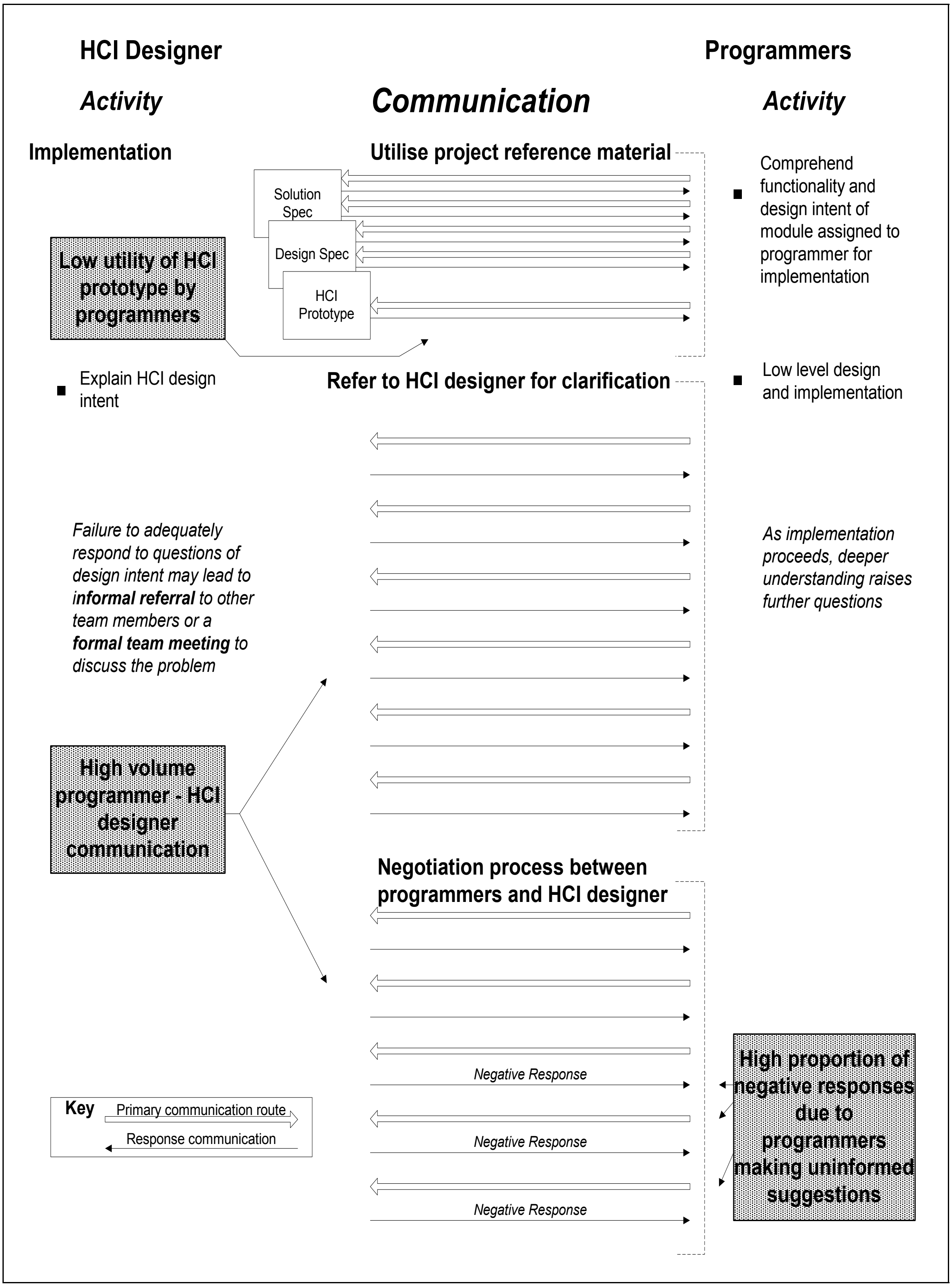

HCI Designer
Programmers
Activity
Communication
Activity
Implementation
Utilise project reference material
Solution Spec
Design Spec
HCI Prototype
■ Comprehend functionality and design intent of module assigned to programmer for implementation
Low utility of HCI prototype by programmers
■ Explain HCI design intent
Refer to HCI designer for clarification
■ Low level design and implementation
Failure to adequately respond to questions of design intent may lead to informal referral to other team members or a formal team meeting to discuss the problem
As implementation proceeds, deeper understanding raises further questions
High volume programmer - HCI designer communication
Negotiation process between programmers and HCI designer
Negative Response
Negative Response
Negative Response
High proportion of negative responses due to programmers making uninformed suggestions
Key
Primary communication route
Response communication

Figure 2.11 Volume of communication between HCI designer and programmer during the implementation phase

2.9.3.6 Specific Findings from Selective Observation of the Interaction Between the HCI designer and the programmers

This section describes specific findings from re-focusing the investigation on the interaction between the HCI designer and the programmers. They are based on the researcher's field experience relating to aspects of the interaction between the HCI designer and programmers on both project 1 and project 2. These findings were validated and supplemented through continuing involvement of the researcher in project 2 during this analysis. Where the findings are supported by earlier investigation, cross-references direct the reader to these.

2.9.3.6.1 Misunderstanding or Misinterpretation of HCI Design Intent

Misunderstandings and misinterpretation of the HCI design intent by the programmers can have many causes, including:

- ambiguity of HCI prototypes (e.g. see 'Software Prototypes' category of section 2.9.2.4.2);

- inaccessibility of the HCI prototype (e.g. see 'Software Prototypes' category of section 2.9.2.4.2);

- inadequacy of the specification and other representations to convey HCI design intent and other definitions of functionality (e.g. see 'Specification' category in sections 2.9.1.6.6 and 2.9.2.4.6);

- lack of explanation of design rationale (e.g. see 'Software Prototypes' category of section 2.9.2.4.2);

- limitations of individual programmers' and HCI designers' abilities, such as the ability to conceptualise (e.g. see 'Visualisation' category in section 2.9.1.6.4 and 'General Project Understanding' category in section 2.9.1.6.5).

The effects of such misunderstandings and misinterpretation can be serious and include:

- misdirected implementation can require major rework;

- misunderstandings and misinterpretation can affect the day-to-day decisions made by the programmers and therefore become ingrained in the software being produced (e.g. see 'General Project Understanding' category in section 2.9.1.6.5);

- failure to understand the importance of a design element may cause programmers to give it a low implementation priority, increasing the risk that it will be left out of the implementation as project deadlines approach (e.g. see data sample in 'Disciplinary Background' category of section 2.9.2.4.1 relating to the problems associated with implementing modeless dialogs);

- misunderstandings may delay the discovery of problems which necessitate redesign at a later stage in the project;

- where team members fail to understand a design concept or fail to understand the importance of a design concept, lengthy team meetings are required to clarify design intent (e.g. see 'Visualisation' category of section 2.9.2.4.4).

Thus, misunderstandings and misinterpretations of HCI design intent by programmers can have many potential causes in the current situation and a number of serious effects.

2.9.3.6.2 Failure to Maintain Conceptual Integrity

If each team member's conceptualisation of the software is different, this is considered a failure to maintain conceptual integrity. Causes of such a failure are similar to those cited as causes of misunderstandings and misinterpretation of HCI design intent but in addition, include:

- uneven distribution of general project understanding and domain knowledge within the team (e.g. see the 'General Project Understanding' category in sections 2.9.1.6.5 and 2.9.2.4.5). For example, some programmers are very focused on their implementation assignment and care little for the general objectives of the project;

- poor communication within the team (e.g. see data sample in 'personality category' in section 2.9.1.6.1);

- changing composition of team causing a loss to team knowledge (e.g. see 'Changing Composition of Team' category in sections 2.9.1.6.7 and 2.9.2.4.7);

- changing requirements inadequately conveyed (e.g. see 'Changes in Requirements' category in section 2.9.2.4.5).

The effects of failing to maintain a good level of conceptual integrity within the team can include any of the problems caused by misunderstanding and misinterpretation. If conceptual integrity is not maintained, the software will be pulled in different directions by those producing it, impacting on the quality and the usability of the software produced.

2.9.3.6.3 Volume of Communication between HCI Designer and the Programmers

As can be seen from figures 2.9, 2.10 and 2.11, the current situation requires the HCI designer to communicate extensively with the programmers throughout several key project phases but primarily the implementation phase. The inadequacy of project documentation (e.g. see the 'Specification' category in section 2.9.2.4.6), the ambiguity and inaccessibility of the prototype (e.g. see the 'Software Prototypes' category in section 2.9.2.4.2), and the lack of recorded design rationale (e.g. see the 'Software Prototypes' category in section 2.9.2.4.2), leave programmers few alternatives to asking the HCI designer. Their motivations for so doing probably have more to do with the fact that the HCI designer has been working on the project for several months longer than most of the programmers, than out of an understanding of the HCI role. Whatever the cause, the HCI designer is required to enable the programmers to gain an understanding of detailed aspects of the proposed software (e.g. see the ‘Specification’ category in section 2.9.2.4.6). As the HCI designer is often outnumbered by programmers, the volume of this communication is considerable. It is also disruptive to the HCI designer’s work. If the information already recorded about the project up until that point were accessible to programmers, this level of communication would be significantly reduced.

2.9.3.6.4 Ineffective Negotiation Between HCI Designer and Programmer

In the current situation, the programmers are not provided with any information (other than verbal) about the rationale for the interface. This could cause them to spend a considerable amount of time and effort implementing a difficult aspect of the user interface (say, reflected in the prototype) which is relatively unimportant. Alternatively, they could omit or modify an aspect of the user interface and cause severe violations to HCI design intent (an example of this from project 2 was apparent from programmers’ attitude to the use of modeless dialogs, which they considered an embellishment on the design but which were actually of fundamental importance to the user’s task – see data sample in the ‘Disciplinary Background’ category of section 2.9.2.4.1). In the communication between programmer and HCI designer, another manifestation of this problem was programmers’ suggested alternative implementation approaches, which were obviously inappropriate and generated a negative response (as shown in figure 2.11). This was almost the best case scenario in the current situation, as this was far better than programmers making assumptions and continuing with the implementation, rather than suggesting alternatives.

2.9.3.6.5 Duplication of Effort

Another symptom of failing to convey HCI design rationale is that programmers are not aware of how much thought has gone into various aspects of the designed user interface reflected in the prototype. Some aspects of the user interface design would have been subject to a lot of design consideration (e.g. the use of modeless dialogs described in the data sample in the ‘Disciplinary Background’ category of section 2.9.2.4.1). Other aspects of the user interface may have only been included in the prototype to fill space or to complete the picture being presented, rather than forming a part of the design (e.g. the ‘Interdisciplinary Issues’ category of section 2.9.1.6.2 describes a prototype demo where I could not dissuade a programmer from focusing on incidental sample data in a prototype). The ambiguity of the prototypes and the lack of design rationale led the programmers to assume that all of the design reflected in the prototype was provisional and not very well thought out. This was often not true and caused programmers to duplicate much of the HCI designer’s considerable effort.

2.9.3.6.6 Low Utility of HCI Prototype by Programmers

The HCI prototype embodied much of the HCI design intent of the software under construction and had the potential to answer many of the questions that programmers asked the HCI designer. This could be seen from a data sample in 'Software Prototypes' category of section 2.9.2.4.2 relating to project 2, which highlights the fact that the HCI designer often utilised the prototype to explain HCI design intent to programmers. However, the utility of the HCI prototype directly by the programmers was low. This was thought to be partly because the prototype was of the chauffeur-driven variety (see section 1.11.3.3.1), making it inaccessible to those without the specialised knowledge required to operate it, or the story accompanying it. The utility of the prototype is also likely to be low because of its ambiguity (see the data sample under the 'Interdisciplinary Issues' category of section 2.9.1.6.1).

Therefore, although prototypes appear to have great potential for conveying HCI design intent to programmers because they are a concretisation of the HCI designer's conceptual model of the proposed software, in current practice, programmers seem to rarely refer to prototypes directly.

2.10 Final Discussion and Conclusions

This section begins with a review of the strengths and weaknesses of the methodology used in this qualitative investigation. Then findings from the descriptive, focused and selective stages of the 'funnel' utilised in the study are summarised. Improvements required to the current situation are identified; possible solutions are proposed and the best option is selected as the future focus of this research. Finally, assuming the proposed solution was in place, the likely future interaction between the HCI designer and programmers is described.

2.10.1 Strengths and Weaknesses of the Study

The strengths and weaknesses of this qualitative investigation are considered in the context of the data obtained (the general strengths and weaknesses of the ethnographic research strategy are covered in section 2.5.2.2).

Strengths

1. The investigation facilitated the development of theory

The starting point for this study was the premise that HCI designers rarely act as a full team-member in a software development team. Therefore, there was no existing theory on which to base preliminary hypotheses relating to the study of this new role. Therefore, the ethnographic research strategy provided an approach to investigating the effects of introducing an HCI designer into a software team in a way that allowed theory to be developed inductively from the data.

It is unavoidable to enter a study without some hint of unspoken hypotheses in the researcher's mind. However, it is believed that the researcher's two-year participant-observation performing the role of HCI designer, of a commercial software team, was a sufficiently long exposure in the 'field' to wipe-away any pre-conceived hypotheses.

2. The investigation exploited the flexibility offered by an ethnographic approach

The ethnographic approach acknowledges the intertwining of data collection and analysis. An extension of this is that the approach encourages flexibility in research design as analysis evolves. The funnel technique was utilised in this study to gradually focus the analysis over the two-year investigation. This proved to be a powerful technique for managing research involvement in a fluid commercial setting, exploiting research opportunities as they arose. For example, following the first year-long study, the opportunity to become involved in a second software project was considered, and ultimately formed the second stage focusing of the funnel.

From the outset of this study, the ethnographic strategy provided the flexibility to fully exploit the research opportunity that was presented by the researcher's Postgraduate

Training Partnership scheme. From the outset, the researcher was placed in a software development subsidiary of an Independent Research Organisation, as well as being enrolled as a Cranfield University student. By taking a flexible approach to the research design, a number of typically difficult challenges in ethnographic research were solved. Thus, having joined a particular software company, problems associated with entering the field and sampling were immediately solved. Similarly, because the researcher joined the software team as a new employee to do HCI design, the 'cover story' for a complete-participant role was very convincing. In other words, the researcher was perceived by employees from the software company as a new employee with part-time study commitments, rather than as somebody studying them.

The flexible approach offered by an ethnographic strategy provided a means to address construct validity concerns by utilisation of multiple sources of evidence. Although participant-observation was the main source of data, ethnographic interviewing provided a method of following up on observations made in an informal way (i.e. in a manner that would not be perceived as research by participants in the setting). Document analysis was also utilised in various ways, e.g. in analysing interactions surrounding the specification documents.

The flexible research strategy also provided a means to address internal validity concerns. Analysis of raw data relating to project 1 was triangulated with two peer analysts using content analysis techniques. Thus, the validity of the first-stage of interpretation of data relating to project 1 was addressed.

3. The ethnographic strategy enabled a natural setting to be studied

The field of HCI is one in which existing theory is much criticised because of its lack of relevance to commercial software practices (see section 1.9). Findings from this study have come from investigation of HCI in practice in a commercial setting. Thus, theories emerging and conclusions drawn have direct relevance to at least one particular commercial software development setting.

The naturalness of the study and the complete-participant role of the researcher minimised reactive effects from the team studied. The team regarded me as an HCI designer rather than a researcher, so I believe reactive effects to have had a negligible effect on the internal validity of the study.

It is important to note that although a compete-participant role was adopted for the participant-observation, this was not covert research. The team were well aware of my research activities but seemed to regard them as my own personal part-time studies, rather than a study of themselves.

4. The skills of the researcher (human instrument)

With any ethnographic investigation, the characteristics, skills and background of the researcher, or human instrument, clearly have an inevitable effect on the data collected and theory developed. In this investigation, my strong software background clearly helped my observation of the setting and analysis of technical documentation. In particular, knowledge of the language, history (and humour) of software engineering helped me to understand what I was observing in the setting.

5. The in-depth study of a particular software development setting

As with most ethnographic studies, this investigation has taken an in-depth look at a particular case. This was an advantageous approach in the context of the research question, where the aim was to investigate the introduction of the new role of HCI designer into a commercial software team. The investigation has generated revelatory results highlighting the real effects of introducing this new role, over a two-year period. Aspects of existing literature support some of the findings from this investigation, suggesting that results may have some external validity but this study did not set out to formally demonstrate generalisable results. Instead, the analysis of this particular case was intended to provide revealing results, the general relevance of which would need to be assessed through further studies.

Weaknesses

1. Limited potential for generalisable findings

The focus of this investigation on a particular case does not provide evidence for the general applicability of results found.

2. Unusual participant role adopted and lack of emphasis on reflection

During this investigation, the researcher took on the role of complete participant in the software team studied. An unusual aspect of this participation was that the researcher's role was a major feature of the study.

In fact, the viability of this study hinged on the researcher performing the role of HCI designer in the team, in order to create a setting to study. The rarity of this role in commercial software development was such that this was considered to be the only way of investigating the effects of introducing this new role.

During the latter stages of writing up the investigation, I realised that my participatory role was so close to the main focus of the study, that I could have placed a greater emphasis on recording and analysing my own reflections. Although introspection does feature in the data collection and analysis, I regret that I did not initially recognise the great potential of this in the context of the study, perhaps because I thought it to be unscientific.

On the surface, a more appropriate role for a participant studying the effects of introducing an HCI designer into a commercial software team would appear to be one that was away from the main focus of the study. Thus, the interaction between the HCI designer and the programmer, for example, could perhaps have been analysed by a researcher performing a different role in the team (i.e. aiming to be a 'fly on the wall'). This would prevent the investigation being coloured by a researcher's participation as one of the key players in the team. However, with the benefit of hindsight, it is precisely the benefits afforded by introspection that led me to conclude that the participatory role adopted by the researcher in this study was appropriate.

Thus, if a similar study were carried out in the future, the researcher would be encouraged to take on the role of HCI designer (or perhaps programmer) but to place more emphasis on introspection of their role and the setting.

3. **Confidentiality and ethical concerns limited the data samples that could be reported**

Because of the paucity of practical HCI research, it was important for the thesis to be suitable for the public domain, rather than treated as confidential. This requirement has restricted the nature and number of data samples that could be reported in the thesis. Because the identity of the host company for the study could not be realistically withheld, and because of the relatively small size of the software development team, coding was an ineffective means of preserving the identity of individuals studied. This has meant that many data samples are reported as general phenomena that occurred, rather than as specific instances.

It is difficult to see how these confidentiality and ethical concerns could be overcome in reporting this kind of study in a way which provides specific contextual examples of phenomena and yet maintained anonymity of people in the setting. At best, data samples of specific events could be recognised by members of the development team reading the thesis, even if other readers would have not been able to identify individuals.

2.10.2 Summary of Findings from the Qualitative Investigation

This investigation began with the research question, 'What are the effects of introducing an HCI designer into commercial software projects as a full team member?' Positive directions described in literature for addressing the realities of commercial software production motivated this question. The paucity of existing research on key areas to be explored by the research question led to the rejection of an *a priori* research design. Instead, an exploratory design was needed. Because HCI theory is often of questionable use to commercial HCI practice, the study had to be grounded in the 'real world'. Investigation of qualitative research strategies led to the selection of an ethnographic strategy for this investigation. Such a flexible approach facilitated the development of theory grounded in the 'real world'. However, the main benefit of the ethnographic strategy was that it created the case to be studied. The researcher joined a typical software development company as a full-time HCI designer for their main software team. The research progressed as an ethnographic study with the researcher adopting the role of complete-participant in the setting. Had this strategy not been adopted, it is believed that an appropriate case or setting to study would not have been found.

Typical participant-observation data collection and analysis methods were adopted, with the 'funnel' technique providing a means of focusing the investigation. The postgraduate scheme that the researcher was enrolled on necessitated their placement in an Independent Research Organisation. In this case, this placement was in a software company. Thus, conducting an ethnographic investigation into the introduction of an HCI designer into a commercial software setting was a natural extension to the research opportunity. Thus, the scheme dealt with a number of typical difficulties associated with ethnographic investigation, in particular, finding a case to study, and entering the field.

Ethnographic investigation of two year-long software projects was conducted, and gradually focused using the funnel technique. The first stage of the funnel was descriptive observations of the project 1 setting following the introduction of an HCI designer into the software team.

The main result of the ethnographic investigation of project 1 was the development of the Venn Diagram conceptual model, an abstract representation of the main 'categories of influence' identified. This model could be considered an emergent 'theory' from the raw data describing the setting (see section 2.5.3.3 - Burgess, 1984). Findings from the project 1 investigation were then written-up using the framework provided by the model.

The Venn Diagram conceptual model provided the focused observations stage of the funnel with its starting point. Thus, the focus of investigation for project 2 was guided by the Venn Diagram conceptual model. Findings from the project 2 investigation were also written-up using the framework provided by this model.

Following the write-up of project 2, a basic comparison of the two projects was carried out. The main observation from this was that project 2 had progressed much more smoothly than project 1. The team had apparently maintained a more consistent understanding of the software under construction in project 2. One reason for this was thought to be that visual prototypes were used to explain the proposed software to team members to a greater extent than had been the case in project 1.

During the development of project 2, developing a visual prototype appeared to be essential to gain an accurate understanding of the user and client requirements for the software. The subsequent use of prototype within the software team was found to be a very powerful means of improving collaboration between an HCI designer and programmers (and other technical team members**). The primary reason for this is believed to be that prototypes addressed fundamental comprehension problems which appeared to be inherent features of the software projects (for example, mixed ability teams).** In the hands of the HCI designer, the prototype was used to help programmers visualise and comprehend the software under production. Perhaps more importantly, using a prototype in this way enabled the team to share a mutually aligned understanding of the software.

During the focused observations of project 2, the researcher re-considered the Venn Diagram conceptual model and concluded that the majority of 'categories of influence' in the model had a comprehension dimension. This comprehension dimension appeared to split 'categories of influence' into fundamental features (almost facts) of the commercial software development setting that created comprehension difficulties, and aspects of the setting designed to improve or facilitate comprehension within the team. Those that were considered **immutable features of the setting** are listed below:

- individual team members' characteristics appear to effect their ability to understand, visualise and explain software;
- software teams appear to be almost inherently mixed ability;
- the changing membership of the team creates a comprehension burden;
- team members from different disciplines experience difficulties understanding each other;
- requirements changes introduce scope for misunderstanding within the team;
- specification documents are an inevitable compromise due to their diverse uses and readerships, but are considered essential milestones.

All but one of the findings from this study that are considered immutable features of the setting have confirmed similar findings from other research described in literature. The exception is the finding that requirements changes introduce scope for misunderstanding, which is believe to be an original, if unsurprising, result.

Aspects of the setting, which apparently aimed to facilitate or improve comprehension within the team, were:

- software engineering representations;
- software specifications;
- providing team members with a general project understanding;
- representations to facilitate software visualisation within the team;
- prototypes and design rationale.

Findings from this study that related to aspects of the setting designed to facilitate comprehension have received less attention in existing literature. However, **some facets of these 'categories of influence' confirm other results reported in literature.** They are listed below.

- Prototyping appears to be an effective means to facilitate visualisation, comprehension and provide a common reference to members of a software team;

- prototypes can be an effective means of helping people to understand what is contained in a written specification;

- design rationale can be stored mainly in the head of the HCI designer and is not conveyed by artifacts from the design process, like the prototype.

However, **many findings from this study, relating to the aspects of the setting that intend to facilitate comprehension, appear to be original.** They are listed below.

- The need for all team members to share a general understanding of the wider issues surrounding the software project, because it is apparent that technical decisions are often made by team members without such knowledge;

- the discovery that sometimes team members believed that a general understanding about the software project was of no use to them;

- that scenarios are complementary to prototyping, as an effective means of helping all members of the team (and others) to visualise the proposed software;

- that prototypes were inaccessible to team members because of their design and development for chauffeur-driven operation;

- that prototypes were sometimes ambiguous and open to interpretation (and misinterpretation) by team members.

The aspects of the setting aimed to facilitate comprehension within the team presented targets for its improvement. Certainly, they were believed to have considerably more potential for change than the apparently immutable features of the commercial software development setting that appeared to cause many of the comprehension difficulties. Furthermore, because this study had apparently made new discoveries in this area, this became the main focus for the following work.

The final focusing of the analysis from the ‘focused observations’ stage of the funnel was directed by the aspects of the setting, which presented targets for improvement. All of these targets had a central position in the interaction between the HCI designer and the programmers, and specifically, how they shared an understanding of the design. Thus, the final focus of this funnel stage was the interaction between these roles and the potential comprehension difficulties that existed between them.

Typical HCI techniques learned from performing the HCI designer role as a participant in the field were used to investigate the interaction between the HCI designer and the programmers in the ‘selective observations’ funnel stage. Firstly, the investigation described the roles, responsibilities and objectives of the HCI designer and the programmer. Then, the nature of the communication and comprehension between these roles across all project phases was described. Observing that the interaction between the roles was at its most intense during the implementation stage, the study focused in greater detail on the nature and volume of communication and comprehension at this stage. Finally, based on ethnographic observation and long field experience, the specific findings relating to the interaction between the HCI design and the programmers were identified and described. These findings were:

Programmers misunderstanding or misinterpreting HCI design intent
caused by:

- ambiguity and inaccessibility of prototypes;
- inadequacy of specifications and other representations;
- lack of explanation of design rationale;
- limitations of individuals' abilities to conceptualise.

resulting in:

- misdirected implementation requiring major rework;
- inappropriate day-to-day decisions made by programmers;
- programmers failing to appreciate the design importance of a particular feature, risking its omission from the implementation;
- misunderstandings delaying the discovery of problems with a design, that create the need for re-design at a later stage;
- lengthy team meetings may be required to clarify concepts.

Failure to maintain conceptual integrity
- causes and effects as in the previous section, plus the following.
caused by:

- uneven distribution of general project understanding and domain knowledge;
- poor communication within the team;
- changing composition of the team;
- inadequately conveyed changes to requirements.

resulting in:

- software is pulled in different directions by those producing it, affecting its quality and usability.

Heavy Volume of Communication between HCI Designer and the Programmers
caused by:

- inadequate project documentation;
- inaccessibility and ambiguity of prototypes;
- lack of recorded design rationale.

resulting in:

- programmers having little alternative but to ask many questions of the HCI designer;
- disruption to the HCI designers work.

Ineffective Negotiation Between HCI Designer and Programmer
caused by:

- lack of recorded design rationale.

resulting in:

- programmers may expend considerable effort implementing an aspect of the system which is faithful to the precise design shown in the prototype, but which actually has only minor importance in terms of the HCI design intent;
- programmers may fail to understand that an apparently trivial aspect of the design shown in the prototype has a fundamental importance to HCI design intent;
- programmers may make alternative design suggestions which are obviously inappropriate and have no basis for negotiation with the HCI designer;
- programmers may make inappropriate assumptions about how the design should be changed in order to better suit the implementation tools/technology available.

Duplication of Effort

caused by:

- lack of recorded design rationale;
- ambiguity of prototypes.

resulting in:

- programmers duplicating the efforts of the HCI designer, replicating their design work because they do not know the level of design consideration that has gone into any given element of the prototype.

Low Utility of HCI Prototype by Programmers

caused by:

- inaccessibility of the prototype due to its chauffeur-driven design – designed to be operated by the person that created it;
- inaccessibility and ambiguity of the prototype for people without knowledge of the story accompanying it.

resulting in:

- unfulfilled potential of prototypes for conveying HCI design intent to programmers.

Many aspects of these findings relate back to earlier, more general observations from the setting. The majority of the findings were related to the use of the visual prototypes in the interaction between HCI designer and programmers. In particular, it was apparent that visual prototypes had a great potential for sharing an understanding of HCI design intent within the team. However, this potential was not full exploited, particularly as the programmers found the prototype inaccessible. Furthermore, the use of prototypes as a means for sharing an understanding of HCI design intent brought with it further problems. For example, the lack of explicit rationale for the design shown in the prototype, and the apparent ease with which aspects of the prototype could be misunderstood or misinterpreted.

The following section considers what can be done to tackle the problems identified in the interaction between the HCI designer and programmer in sharing a common understanding of HCI design intent.

2.10.3 Improvements Required in the Current Situation

This section outlines the improvements that are required (and have scope) in the problem areas identified at the final focusing of the 'funnel' (i.e. specific findings from the selective observations reported in section 2.9.3.6). Improvements are required to:

- reduce the misunderstanding and misinterpretation of HCI design intent;
- improve the maintenance of conceptual integrity within the team;
- reduce the volume of enquiries that programmers make of the HCI designer;
- improve the programmers' understanding of the design rationale underlying the user interface to put them into a more realistic position in negotiation with HCI designers, and to enable them to suggest reasonable design alternatives;
- reduce the duplication of design effort by better explaining design work that has gone before;
- increase the utility of the concretisation of the HCI designer's mental model which is available from the HCI prototype.

2.10.4 Potential Solutions

This section describes potential solutions to the problem areas in the collaboration between the HCI designer and programmers.

2.10.4.1 Improve Selection of Team Members

Reducing the mixture of abilities evident within the software team by selection, to achieve a consistently high skill level, may reduce some of the collaboration problems. In a highly skilled team, communication could be improved through a deeper shared understanding of the technology, as well as general project objectives and domain knowledge.

This solution does not acknowledge the commercial reality that programmer selection is difficult (see section 1.10.2.1). Neither does the approach acknowledge that when a mediocre programmer has been recruited, it is more difficult to subsequently terminate their contract than it is to reassign them to another project.

2.10.4.2 Improve Project Documentation, Primarily the Specification

In general, there is considerable scope for improvement of project documentation. The specification is usually the best example of a document which has the greatest potential to improve, particularly with regard to presentation and clarity.

However, although such improvements to specifications could be advantageous, it is apparently a feature of commercial software development that the specification document, which the team members use as a working document, also has a multitude of other uses, including forming a contract with the client. This diversity of uses means that the specifications are often a compromise (see section 2.9.2.7.6).

2.10.4.3 Maintain Development Team throughout Project

Maintaining the same team members throughout the duration of a software project would be a way of maintaining project knowledge in the team. This would reduce the overall communication burden caused by people leaving the team and new people joining.

Scarcity of resources is a commercial fact which dictates that team members have to be utilised in their most productive roles. It would often not be commercially viable to involve all programmers at the very start of a project.

2.10.4.4 Improve the Usability of HCI Prototypes

It is partly the chauffeur-driven nature of HCI prototypes, requiring the user to have specialist knowledge to operate them, that causes their inaccessibility. One means of addressing this problem would be to improve the usability of such prototypes. Constructing prototypes which can be tried out 'hands-on' could improve their usability, as could adding 'help' features and providing other documentation.

Commercial realities mean that HCI prototypes are often produced rapidly. Devoting time to improving the usability of the prototype and documenting the best way to demonstrate it may prove to be expenses that are hard to justify.

2.10.4.5 Adopt an Evolutionary Approach to Development

Perhaps the most radical means of improving the collaboration between the HCI designer and the programmer would be to change its nature. Instead of designing the user interface in its entirety before the majority of the programmers join the development team, the HCI designer and the programmers could collaborate, gradually evolving the software to meet the user and client requirements.

Although this approach has considerable merit from an HCI perspective, it has been criticised from a programming and IT perspective. Claims are made that software produced in this way is not robust (see sections 1.7.2 and 1.3.2). Even if such difficulties could be addressed, the commercial reality is that evolutionary developments require a special set of conditions. One fundamental problem with the approach is that fixed price contracts are still dominant in the software industry (see section 1.3.2).

2.10.4.6 Education about the Project

Education of all team members about general project objectives and the application domain could improve the communication and negotiation processes within the team.

Taking time to formally prepare education material for team members who are expected to find out about the project for themselves is unlikely to gain management approval.

2.10.4.7 Improve Training

Training may be of some use in improving teamworking and communication skills.

However, programmers have been found to be an occupational group with extremely low needs for social contact at work (see section 1.10.2), so team working and communication skills training may be of only limited use.

2.10.4.8 Improve the Representation and Communication of Changes

Changes in requirements and subsequent changes to design which occur throughout the project could be better represented and communicated.

The number of changes which can occur, the multiple sources of changes and the predominantly paper document-based approach to software production makes effective representation and communication of changes inherently difficult.

2.10.4.9 Recording Design Rationale

Explicit records of design rationale could provide a better basis for programmers to negotiate implementation alternatives with the HCI designer.

Researchers are undecided as to whether the cost of producing records of design rationale is justifiable (see section 1.11.3.4). One concern is that design rationale representations can be too abstract; and linking them to an exemplar system would provide a better explanation.

2.10.4.10 Utilise a Prototype-Centred Explanation Tool

Elements of many of the most positive solution options suggested above (i.e. in sections 2.10.4.2, 2.10.4.4, 2.10.4.6, 2.10.4.8 and 2.10.4.9) could be incorporated into one explanation tool centred around probably the best means of facilitating comprehension with the team - the chauffeur-driven HCI prototype. This is considered of particular relevance because such prototypes are routinely created during a software project, and therefore represent an existing resource which could be further exploited.

The concept of a prototype-centred explanation tool was conceived by the researcher as a possible 'add-on' to any chauffeur-driven prototype. The add-on should automatically provide guided tours of the chauffeur-driven prototype without the need for the original chauffeur. This should increase the accessibility of the prototype to programmers and other team members and improve the utility of the prototype.

In addition, the prototype-centred explanation should also provide information about each aspect of the prototype. Exactly what is written about each aspect of the software reflected in the prototype will depend on the HCI designer creating the explanation, but this should be flexible. The key benefit is that this add-on becomes an information area which allows the HCI designer to jot down, in context, any other information available that supports the design shown in the prototype. If for example, an aspect of the design reflected in the prototype had been subject to a great deal of design consideration and this was not apparent from the appearance of the prototype, it could be described in more detail in the associated information area. Conversely, where aspects of the design represented in the prototype are ill thought out (e.g. non-essential features of the design), the information area allows the HCI designer to convey this.

Although the information area is free format, allowing the HCI designer to write about aspects of the prototype and include pictures and diagrams if desired, some information content would be suggested by the prototype-centred explanation tool. To achieve this, empty slots for certain types of information would appear within each information area. The most important of these slots would be 'design rationale', prompting the HCI designer to explain the rationale underlying each aspect of the design shown in the prototype which they explain. Others might include:

- a 'see also' slot where HCI designers would be encouraged to cross-reference the information in the prototype-centred explanation;

- a 'general project information' slot enabling project objectives and an overview of the domain to be described;

- a 'users and tasks' slot to describe the users and their tasks and to animate their likely future use of the proposed software by using scenario-based prototype 'tours'.

The prototype-centred explanation tool concept therefore could provide some level of improvement to each of the problem areas identified in section 2.10.3, incorporating many of the other solution options suggested in section 2.10.4. Furthermore, this solution option does not preclude any of the others suggested that it does not encompass directly (i.e. sections 2.10.4.1, 2.10.4.3, 2.10.4.5 and 2.10.4.7).

By providing programmers with access to the HCI designer's conceptualisation of the proposed software, concretised in the prototype, misunderstandings and misinterpretation of HCI design intent could be reduced and a greater level of conceptual integrity within the team maintained. These potential improvements could be further supported by the provision of general project information and information about users and tasks.

If the prototype-centred explanation tool becomes a successful means of facilitating comprehension and communication within the team, the volume of enquiries addressed to the HCI designer could be reduced.

By explaining design rationale in the context of a prototype tour, the programmers should gain a better understanding of why the HCI design is the way it is. This will enable them to suggest design alternatives, which are consistent with HCI design intent and will put them in a better position to negotiate.

Because prototype-centred explanation tool information areas allow the HCI designer to explain aspects of the prototype, duplication of effort will be reduced. Instead of programmers thinking through all of the issues surrounding a design which the HCI designer has already thoroughly considered, they can take the HCI designers work as a starting point.

If successful, the prototype-centred explanation tool add-on, should by definition, increase the utility of the visual prototype.

The prototype-centred explanation tool concept derives flexibility from its focus on the chauffeured prototype, which is in itself a flexible technique. In this respect, the prototype-centred explanation tool should be very different from failed CASE tools and IPSEs which attempted to structure the work of those using them (see section 1.5.1). Thus, the prototype-centred explanation tool should not enforce any structure on the work practices of those using it, although it could have a marginal influence on the HCI designer's work. This flexibility should also enable the prototype-centred explanation tool to be used within a wide range of software lifecycles and development situations.

From the point of view of commercial viability, a prototype-centred explanation tool would need to prove itself in a similar way to that suggested for recording design rationale (see section 3.4.9). It will need to be demonstrated that the benefits of using a prototype-centred explanation tool representation outweigh the costs associated with its production.

2.10.5 Solution Option Selected as a Focus for the Next Research Stage

The prototype-centred explanation tool concept introduced here incorporates many of the positive measures which could lead to improvements in the communication and sharing of understanding between the HCI designer and programmers. For example, design rationale would be explained alongside an image of the related software, as design rationale researchers have suggested (see section 1.11.3.4). The prototype-centred explanation tool approach is believed to have the most potential for providing a commercially viable, widely applicable and effective means of facilitating and improving the communication and comprehension within the software team, particularly between the HCI designer and programmers.

2.10.5.1 Proposed Future Situation

Based on the analysis of the interaction between HCI designer and programmers described in section 2.9.3, this section illustrates how the future situation could be supported with the hypothesised prototype-centred explanation tool. The tool is expected to contribute to all of the improvements required in the current situation, identified in section 2.10.3.

Sections 2.10.5.1.1 and 2.10.5.1.2 clarify the requirements for the prototype-centred explanation tool by explaining the role it should play in the future situation in supporting the collaboration of the HCI designer and programmers.

2.10.5.1.1 Detailed Analysis of Potential Programmer Utility of a Prototype-Centred Explanation Tool During Implementation Phase

Figure 2.12 is based on the model of the existing communication that occurs between the programmers and the HCI designers in the implementation phase, shown in figure 2.10 and described in section 2.9.3.4. Figure 2.12 is a simplified view of how the programmers could use the information in the prototype-centred explanation tool in this phase (labelled "ProtoTour" in the diagram). The tool could provide the following information to the programmers:

- animated (and narrated) 'tour' of the HCI prototype bringing to life project reference material;

- description of design features, linking to and further explaining other project reference material;

- representation of states in the proposed software via state-transition diagrams (STDs);

- further description of the proposed software's operation, including demonstration of required states utilising prototype 'tour';

- explanation of design intent relating to key design features and identification of background features shown in the HCI prototype;

- explanations of user tasks and their future tasks, supported by prototype scenario-based ‘tour’;

- details about users;

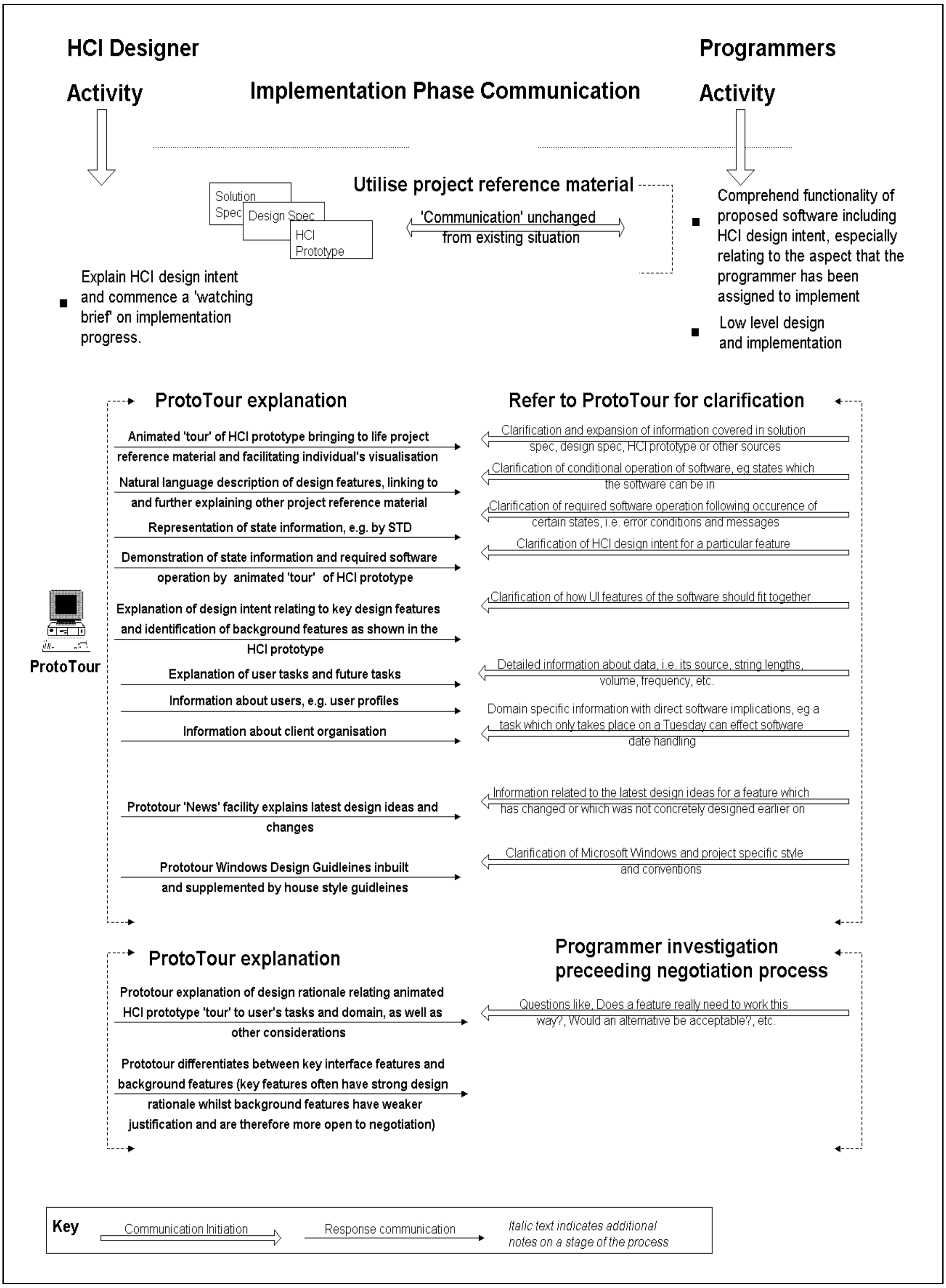
- information about the client organisation and other general project information;

- ‘News’[7] on latest design ideas and changes;

- platform and house style guides.

Information about the design rationale underlying the HCI design shown in the prototype will also form part of the tool’s explanation and justifications may be linked to scenario-based prototype ‘tours’ to explain the rationale. The aim of this is to enable the HCI designer to be explicit about why elements of the software have been designed as they have and which elements of the prototype are fundamental and which are background features. This information is intended to place the programmers in a stronger negotiating position, by providing them with the understanding necessary to suggest realistic alternatives to the HCI designer’s.

[7] The concept of project ‘News’ to convey recent changes to software specifications and design within the software team was cut from the initial tool design and implementation described in chapter 3 to make the scope of implementation and evaluation more feasible. The concept is re-visited in the “future research” section of the final discussion (chapter 5).



Utilise project reference material

'Communication' unchanged from existing situation

- Explain HCI design intent and commence a 'watching brief' on implementation progress.

- Comprehend functionality of proposed software including HCI design intent, especially relating to the aspect that the programmer has been assigned to implement
- Low level design and implementation

ProtoTour explanation

Animated 'tour' of HCI prototype bringing to life project reference material and facilitating individual's visualisation

Natural language description of design features, linking to and further explaining other project reference material

Representation of state information, e.g. by STD

Demonstration of state information and required software operation by animated 'tour' of HCI prototype

Explanation of design intent relating to key design features and identification of background features as shown in the HCI prototype

Explanation of user tasks and future tasks

Information about users, e.g. user profiles

Information about client organisation

Prototour 'News' facility explains latest design ideas and changes

Prototour Windows Design Guidleines inbuilt and supplemented by house style guidleines

Refer to ProtoTour for clarification

Clarification and expansion of information covered in solution spec, design spec, HCI prototype or other sources

Clarification of conditional operation of software, eg states which the software can be in

Clarification of required software operation following occurence of certain states, i.e. error conditions and messages

Clarification of HCI design intent for a particular feature

Clarification of how UI features of the software should fit together

Detailed information about data, i.e. its source, string lengths, volume, frequency, etc.

Domain specific information with direct software implications, eg a task which only takes place on a Tuesday can effect software date handling

Information related to the latest design ideas for a feature which has changed or which was not concretely designed earlier on

Clarification of Microsoft Windows and project specific style and conventions

ProtoTour explanation

Prototour explanation of design rationale relating animated HCI prototype 'tour' to user's tasks and domain, as well as other considerations

Prototour differentiates between key interface features and background features (key features often have strong design rationale whilst background features have weaker justification and are therefore more open to negotiation)

Programmer investigation preceeding negotiation process

Questions like, Does a feature really need to work this way?, Would an alternative be acceptable?, etc.

Italic text indicates additional notes on a stage of the process

Figure 2.12 A simplified view of how programmers' comprehension of software under construction could be facilitated using the prototype-centred explanation tool (labelled 'ProtoTour') in the implementation phase

2.10.5.1.2 An Illustration of the Volume of Communication Between HCI Designer and Programmer During Implementation Phase Following the Introduction of a Prototype-Centred Explanation Tool

Figure 2.13 highlights three key points about the projected communication and comprehension activities within the implementation phase following the introduction of a prototype-centred explanation tool. Firstly, the utility of the HCI prototype is expected to increase dramatically as it is at the heart of the explanation tool. Secondly, the programmers' understanding of the proposed software should be increased by reference to the prototype-centred explanation tool, which is expected to reduce the volume of communication between the HCI designer and programmer. Thirdly, by providing programmers with design rationale information, their understanding of the underlying reasons for the user interface design should put them in a better position to offer sensible alternatives. This is expected to reduce the overall volume of negotiation between HCI designer and programmer. At the same time, it should improve the quality and relevance of alternatives suggested by the programmers.

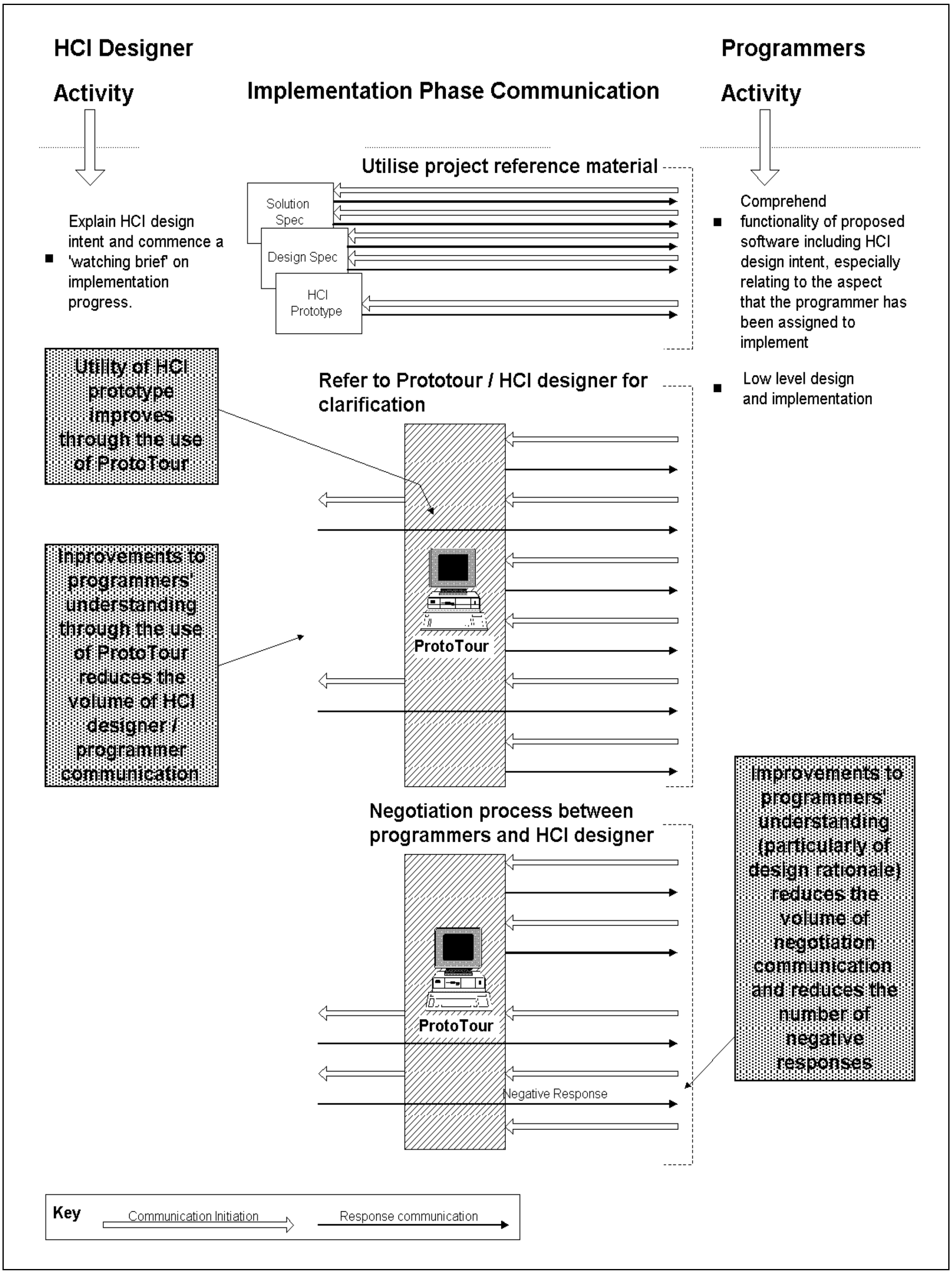

HCI Designer
Activity
Implementation Phase Communication
Programmers
Activity
Utilise project reference material
Solution Spec
Design Spec
HCI Prototype
Explain HCI design intent and commence a 'watching brief' on implementation progress.
Comprehend functionality of proposed software including HCI design intent, especially relating to the aspect that the programmer has been assigned to implement
Low level design and implementation
Refer to Prototour / HCI designer for clarification
Utility of HCI prototype improves through the use of ProtoTour
Improvements to programmers' understanding through the use of ProtoTour reduces the volume of HCI designer / programmer communication
ProtoTour
Negotiation process between programmers and HCI designer
Improvements to programmers' understanding (particularly of design rationale) reduces the volume of negotiation communication and reduces the number of negative responses
ProtoTour
Negative Response
Key
Communication Initiation
Response communication

Figure 2.13 Expected volume of communication between HCI designer and programmer during implementation phase following the introduction of a prototype-centred explanation tool (labelled 'ProtoTour')

2.10.6 The Next Stage in the Research

The next stage of this research focuses on designing and constructing a prototype-centred explanation tool for the purposes of testing the concept. The next chapter describes the design and implementation of such a tool, called ProtoTour and the following chapter describes the experimental evaluation of its utility.

Chapter 3 The Design and Implementation of a Prototype-Centred Explanation Tool - ProtoTour

3.1 Introduction

The qualitative investigation described in chapter 2 found the comprehension of HCI design intent within the software team to be of great importance, particularly in the interaction between the HCI designer and the programmers. Some apparently immutable features of the commercial software setting appeared to create comprehension difficulties, e.g. changing requirements. Other aspects of the setting appeared to be designed to aid comprehension, e.g. software specifications. Of all the aspects designed to aid comprehension, prototyping had the most potential and acknowledged the immutable realities of commercial software development. However, the use of prototypes for facilitating the comprehension of HCI design intent was found to be flawed in a number of ways:

- prototypes could be misunderstood or misinterpreted;

- prototypes could be inaccessible (leading to under-utilisation and a high volume of communication between the HCI designer and the programmers);

- prototypes could not convey design rationale (consequently, they could not provide a basis for negotiation between the HCI designer and the programmers).

A prototype-centred explanation tool is proposed in chapter 2 as a means of exploiting the potential which prototypes were found to have, whilst addressing the key flaws in their use.

This chapter describes the design and implementation of a prototype-centred explanation tool called ‘ProtoTour’. Its primary aims were; to reduce prototype ambiguity; increase prototype accessibility; and provide programmers with additional information to enable them to provide an effective contribution to the design.

3.2 Initial Concepts

This section explains the initial concepts for the ProtoTour representation and describes how the concepts evolved from the findings of the qualitative investigation of chapter 2 and the further analysis of HCI designer and programmer collaboration described in chapter 3.

3.2.1 Automated Prototype 'Tours'

Visual prototypes presenting a façade of a proposed software are often prepared during software development in order to gain a better understanding of the requirements for the proposed software (as described in the literature, see section 1.7.4.1). Such prototypes, which can present a complex façade, can be prepared rapidly and are designed to be presented by the prototype builder. This person is likely to have a deep understanding of the conceptual model of the proposed software for which the visual prototype is merely a concretisation. In order for the visual prototype to accurately reflect the conceptual model, it must be operated by someone who understands not only the conceptual model but also the 'smoke and mirrors' tricks (hidden keystrokes, mouse clicks, etc.), which present the illusion of apparent functionality using only the visual prototype façade.

The HCI designer is in a good position to be the prototype builder. Building the prototype allows the HCI designer to be involved in the early formation of the basic software concepts underlying the design. The user interface is therefore designed first. Prototyping allows the HCI designer to confirm that the basic appearance of the software conforms to the user/clients' understanding of what they want. Requirements can be quickly incorporated into the design and reflected back in the prototype for discussion.

Having evolved the visual prototype to encapsulate requirements for the proposed software, it is used as a communication medium within the development team as the implementation progresses into the design phase. The composition of the team often changes as implementation gains pace. New team members may initially know nothing of the often complex conceptual model of the proposed software which is reflected in the visual prototype and held in the heads of some existing team members. Conceptual models of the software held by individual team members need to be closely aligned if the end product is to be coherent and the process of getting there is to be smooth. The visual prototype often appears to be a good means of visualising the software. If an implementation issue arises, the visual prototype can be used to query the conceptual model of the software (held in the heads of some of the team members). Technical documentation surrounding the implementation rarely provides such a good window on the conceptual model.

A problem arises when the prototype's concretisation of the conceptual model can only be effectively operated by the HCI designer. After dedicating considerable efforts to producing and validating a visual prototype, its accessibility is limited. Because the visual prototype on its own is only a façade, it has some strange behaviour in conventional software terms and relies on the operator's knowledge of hidden 'tricks' to present it correctly. Furthermore, the prototype provides a visualisation which usually fits into a story of how the software is being operated. Thus the prototype's shallow functionality only really makes sense as part of a scenario walkthrough. A programmer with only basic skills and no experience of a particular visual prototype could make various aspects of it apparently 'work', but without the correct sequence of events, knowledge of the hidden 'tricks' and an understanding of the story, the apparent functionality would be nonsensical and misleading.

A further consequence of the limited accessibility is that team members have to ask the HCI designer to operate the visual prototype each time they want to query the model. The bigger the software team the worse this could become. This bottleneck may also dissuade team members from asking the relevant questions, instead leading them to make assumptions.

The ProtoTour concept was invented to provide an automatic means of driving the visual prototype, in effect a tour of the prototype (hence the name). The tour would be created by recording sequences of events from the visual prototype. By arranging these sequences and adding a narrative, sequences of events from the visual prototype could be played back to anybody with an interest in how the proposed software should work. The ProtoTour recordings have the potential to improve on the correctness of the prototype visualisation by editing the recordings. Sequences of events in the prototype which less accurately represent the proposed software can be cut from the recording or altered (in a similar way to film editing procedures and special effects). So, as well as being more accessible, the ProtoTour walkthroughs are potentially more accurate representations of how the proposed software should work.

3.2.2 Focussed Specification

A software specification is still a primary document in a software development (see section's 1.6, 2.9.1.6.6 and 2.9.2.4.6 for evidence). Technical specification documents attempt to describe the required functionality of the software under construction (in detail). Visual prototypes produced prior to this can be turned into a set of screen pictures to accompany technical descriptions of elements of the proposed software. Because the resulting technical documents can be difficult to read, the use of the visual prototype can be continued in order to provide an effective means of communication within the team (see sections 2.9.1.6.2 and 2.9.2.4.2). Furthermore, as the implementation phase gets under way (after the specification has been completed), it may become apparent that some planned functionality is not viable in the way that it has been designed. Returning to the prototype allows team members to test out alternative design ideas and consider their effect on the overall conceptual model of the proposed software. The technical specification document does not facilitate visualisation of the conceptual model as effectively as the prototype. Its role is to provide a more detailed description of the proposed software.

Some parts of the visualisation of the proposed software do not get represented in the specification. For example, HCI design may have produced some complex interactional ideas in the visual prototype which are not effectively described. There may be several reasons for this: they are too subtle; only static printed screen pictures can be included in the specification; the interaction is extremely complex to describe in words; or they may be just forgotten.

The ProtoTour concept was expanded to address some of these problems. After recording a visual prototype sequence, a narrative description is added to the recording to explain how the proposed software will be used. This style of explanation can also be adopted to present an exhaustive explanation of the functionality of particular elements. In other words, the extent of the functionality and interactivity of a particular element is described, as opposed to how the element fits into the bigger picture when the software is being used.

The HCI designer will be forced to carefully consider all aspects of a particular software element (e.g. a window), as they prepare visual prototype walkthroughs and associated narrative (tours - recorded as animation) to describe HCI design intent. When describing the element in a written specification, the description is centred around a static picture. The preparation of a recorded sequence and narrative is likely to require more thought than a written technical description and has better visual prompts as to what needs to be specified. ProtoTour might therefore make specifications of proposed software reflected in visual prototypes more thorough and complete.

Because there is information about proposed software which is not best presented as a narrative, the ProtoTour concept could be expanded to include an on-screen text element. This way, more text-oriented information supports the recorded walkthroughs and vice versa. To optimise the utilisation of on-screen text facilities, hypertext, search mechanisms and colour pictures can also be incorporated into the ProtoTour concept. Researchers believe that such hypermedia facilities have potential as an explanatory and learning medium (see section 1.11.4.3) and this potential was thought to be exploitable within ProtoTour. Such facilities for on-screen explanation are thought to provide a better link to the visualisation and communication potential afforded by a visual prototype than a technical specification.

3.2.3 Design Rationale

When using visual prototypes and specification documents to support software implementation design rationale is not captured and is rarely routinely recorded elsewhere (see section 1.11.3.4 and the ‘Software Prototypes’ category within section 2.9.2.4.2 for evidence). If not recorded, the design rationale is likely to remain in the heads of the designers and not be communicated within the software team.

Without an understanding of rationale underlying the design, programmers would not be in a position to suggest alternative implementation options or negotiate alternatives with the HCI designer. Programmers are usually closer to the technology than other team members, so in terms of technical knowledge, they are in the best position to suggest better ways of implementing aspects of the proposed software. These suggestions may be different from the implementation implied in the visual prototype, but may be significantly easier to implement. Therefore, if the implementation is to make optimal utilisation of technologies available, programmers need to suggest alternatives. To do this, a knowledge of the rationale for the original designs is thought to be essential.

Visual prototypes are inherently misleading with respect to design rationale. Whilst some aspects of a prototype may be the subject of a large amount of design effort and consideration, these co-exist in the prototype illusion alongside other aspects of the design which may have just been added to fill space. The viewer of an expertly presented prototype façade has no clue to the level of design consideration that has gone into the different aspects. This has two effects. Firstly, when the programmers begin to implement an aspect of the software they duplicate some of the thinking that has already been done by someone else in the team. Secondly, programmers may begin to implement an aspect of the visual prototype without giving it much consideration only to discover that the concept was ill-thought out (put in the prototype to fill space) and is seriously flawed.

Design rationale was therefore included as another facet of ProtoTour. Centring an explanation of design rationale on the prototype enables the HCI designer to highlight, both the parts of the design that are well thought out, and those that are not. It should allow them to thoroughly explain the issues surrounding the design and the alternatives considered. This may have several effects. Firstly, the design rationale information should enable programmers to suggest implementation alternatives which acknowledge the constraints of the design. Secondly, when a programmer begins to implement an aspect of the prototype, the extent of the thinking and design consideration that has gone before is available, which should reduce costly duplication of effort. Thirdly, the programmer should gain an appreciation of the HCI designer's reasoning and confidence that important issues have been thought through. Fourthly, as a visual prototype evolves, the ProtoTour representation could evolve with it, creating a record of design decisions made.

3.2.4 General Project Information

When programmers begin work on a software project, it is usual for a considerable amount of analysis and design to have occurred already (for example, in project 2 some programmers joined the software team around six months after the project had started - see section 2.9.2.4.7). Thus, programmers do not usually follow a project through from its inception and therefore, might not have a thorough understanding of project objectives and background. Written specification documents and the visual prototype do not appear to fill in all of the gaps in their understanding; general knowledge about the project often appears to be one such gap. Despite this, during their day-to-day work programmers often made seemingly insignificant decisions, which shaped the implementation (see the 'General Project Understanding' category in section 2.9.2.4.5 for evidence). Such decisions made in isolation can unwittingly steer the software in the wrong direction. Any single bad judgement or misunderstanding guiding the implementation can have very severe consequences; the reversal of which can be extremely costly.

Because general project information was not communicated effectively to programmers, the ProtoTour concept was further expanded to contain a section explaining general project information.

3.2.5 Task and User Information

Much of the basis of design rationale underlying designs shown in the visual prototype comes from the HCI designer's understanding of the users and their tasks. To properly visualise and understand the proposed software, it is often necessary to view it from the perspective of the users. Task and user information was not commonly found to be palatable by implementers; some of whom would rather the HCI designer worried about this aspect of the development and let them know the outcome.

By integrating the user and task information with explanations and walkthroughs of the visual prototype in ProtoTour, the implementers may be led through this information. For example, a full walkthrough of a visual prototype could be presented following a typical use scenario drawn from task analysis. As with general project understanding discussed in section 3.2.4, programmers make minute decisions about the implementation as part of their day-to-day work. The better their understanding and appreciation of the users and tasks, the less likely these decisions should be to unwittingly violate design intent.

3.2.6 Style Guide

Rapidly constructed visual prototypes were often found to gloss over specific implementation details. Such details include the precise specification of the style and layout of elements of the user interface. Attention to these details is considered to be essential for achieving consistency of the user interface with the environment in which the software is working (e.g. Microsoft Windows) and with the company software style.

The ProtoTour concept was therefore further expanded to include links to an on-line user interface style guide. This would enable an on-line house style guide (or general Windows style guide) to be linked with the ProtoTour representation of any visual prototype produced, giving implementers access to detailed and precise user interface style and layout information. Giving the implementers access to such information may also reduce the number of interruptions and queries the HCI designer receives.

3.3 The Design

This section describes the design of the first prototype of a ProtoTour tool.

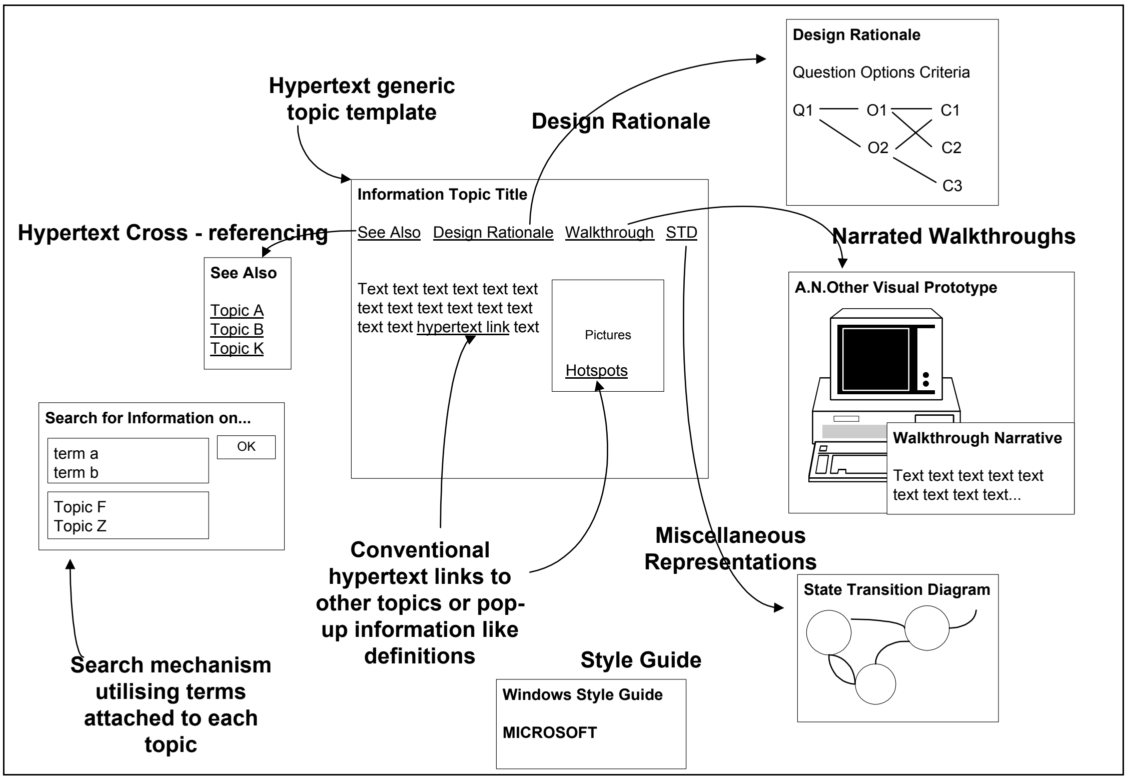
An important underlying philosophy of the ProtoTour concept is that its use should not impose any structure or particular way of working on the users (both authors and readers). The flexibility of visual prototypes is thought to be major factor in their increasing use, so ProtoTour must not restrict this. Administrative burdens placed on software teams with the introduction of tools like Integrated Project Support Environments (IPSEs) have clearly thwarted their uptake (see section 1.5).

Because ProtoTour is designed to 'piggyback' an existing visual prototype, trying to explain the design of it in isolation would make little sense. Therefore, the design of ProtoTour is described in this chapter by illustrating the kinds of information and functionality that ProtoTour would contain if it had been applied to a specific visual prototype example. The example used is that of a visual prototype for proposed software called ELDER (Engineering Line Diagram advisER). The ELDER visual prototype was selected as the example because it was developed as part of a commercial project.

ELDER itself was predominantly hypertext explanation software, like ProtoTour. Thus, the example comprises ProtoTour's hypertext explanation facilities applied to an explanation of the ELDER hypertext explanation software. This was an unfortunate and unavoidable situation, but one which ultimately may have generated more feedback from formative evaluation than would otherwise have been the case (for example, section 3.5 describes the evaluator's difficulty distinguishing ProtoTour from the ELDER prototype).

3.3.1 Overview of ProtoTour

The ProtoTour concept begins with a generic topic template. ProtoTour links together any number of template pages in a hypertext environment but allows the content of each page to be flexible. Furthermore, a template page can be used to activate other features which are not hypertext-based, like the animated walkthrough sequences. The actual content and arrangement of a ProtoTour representation is left to the discretion of the HCI designer. Figure 3.1 outlines the main features of a ProtoTour representation.



Hypertext generic topic template
Design Rationale
Design Rationale
Question Options Criteria
Q1
O1
C1
O2
C2
C3
Information Topic Title
Hypertext Cross - referencing
See Also
Design Rationale
Walkthrough
STD
Narrated Walkthroughs
See Also
Topic A
Topic B
Topic K
Text text text text text text text text text text text text text text hypertext link text
Pictures
Hotspots
A.N.Other Visual Prototype
Walkthrough Narrative
Text text text text text text text text text...
Search for Information on...
term a
term b
OK
Topic F
Topic Z
Conventional hypertext links to other topics or pop-up information like definitions
Miscellaneous Representations
State Transition Diagram
Search mechanism utilising terms attached to each topic
Style Guide
Windows Style Guide
MICROSOFT

Figure 3.1 Overview of the main features of a ProtoTour representation

The remainder of this section will be devoted to explaining all of the elements of ProtoTour in more detail. This will include the main features illustrated in figure 3.1 and will also encompass specific examples of the application of the generic topic template.

3.3.2 The ProtoTour Main Screen

The ProtoTour main screen (see figure 3.2) offers four primary categories of information relating to the proposed software illustrated in the prototype: Project Information; software 'in Action'; Design Intent; and the relevant Style Guide. The primary option is Design Intent as this explains and demonstrates the visual prototype. Software 'in Action' contains information about the users, their tasks and how they will use the proposed software. Project Information describes the objectives, philosophy, budget and timescales for the software project.

At this first screen, the ProtoTour user may immediately utilise the search facility to bypass hypertext navigation and go directly to the information they are interested in.

An 'About ProtoTour' button provides a brief pop-up explanation of what ProtoTour is.

The illustration on the main screen utilising a film show metaphor is designed to have a strong purpose. During formative evaluation (discussed further in section 3.5), evaluators had difficulty distinguishing ProtoTour from the visual prototype example it was explaining. The film show metaphor was intended to reinforce the fact that ProtoTour is presenting the visual prototype.

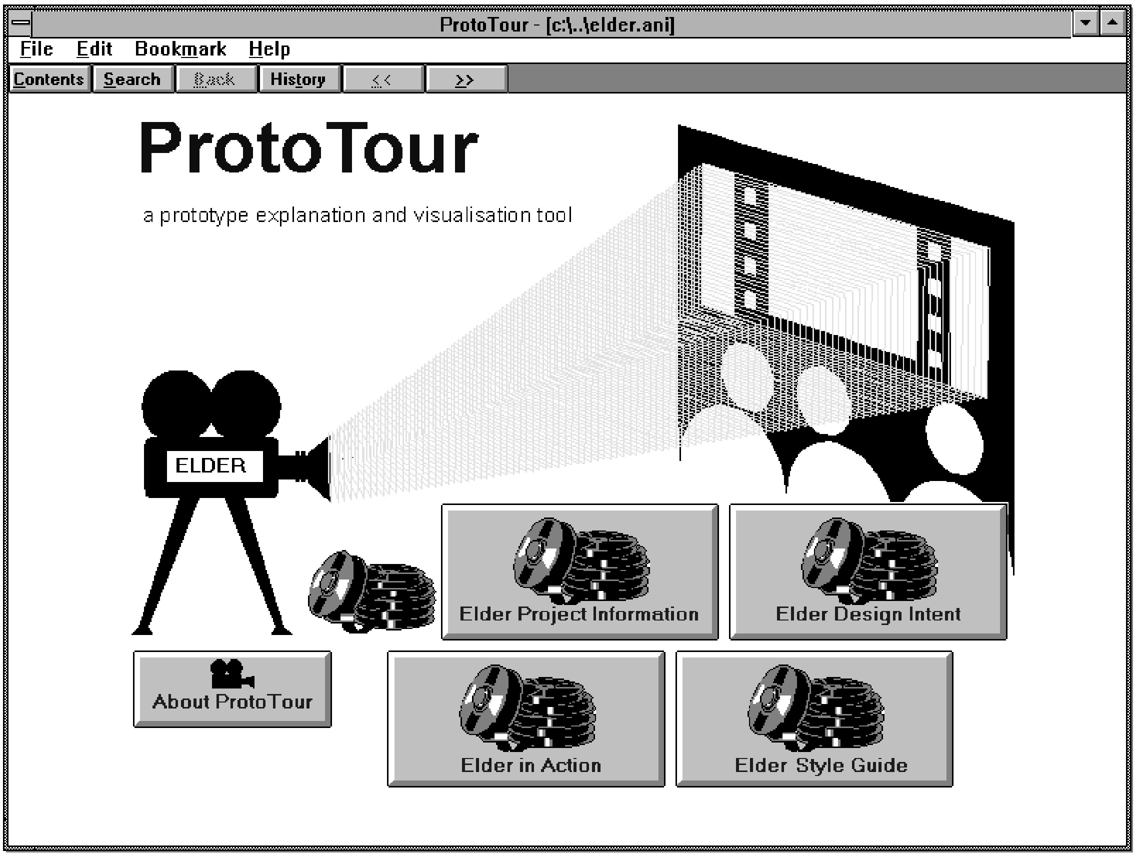
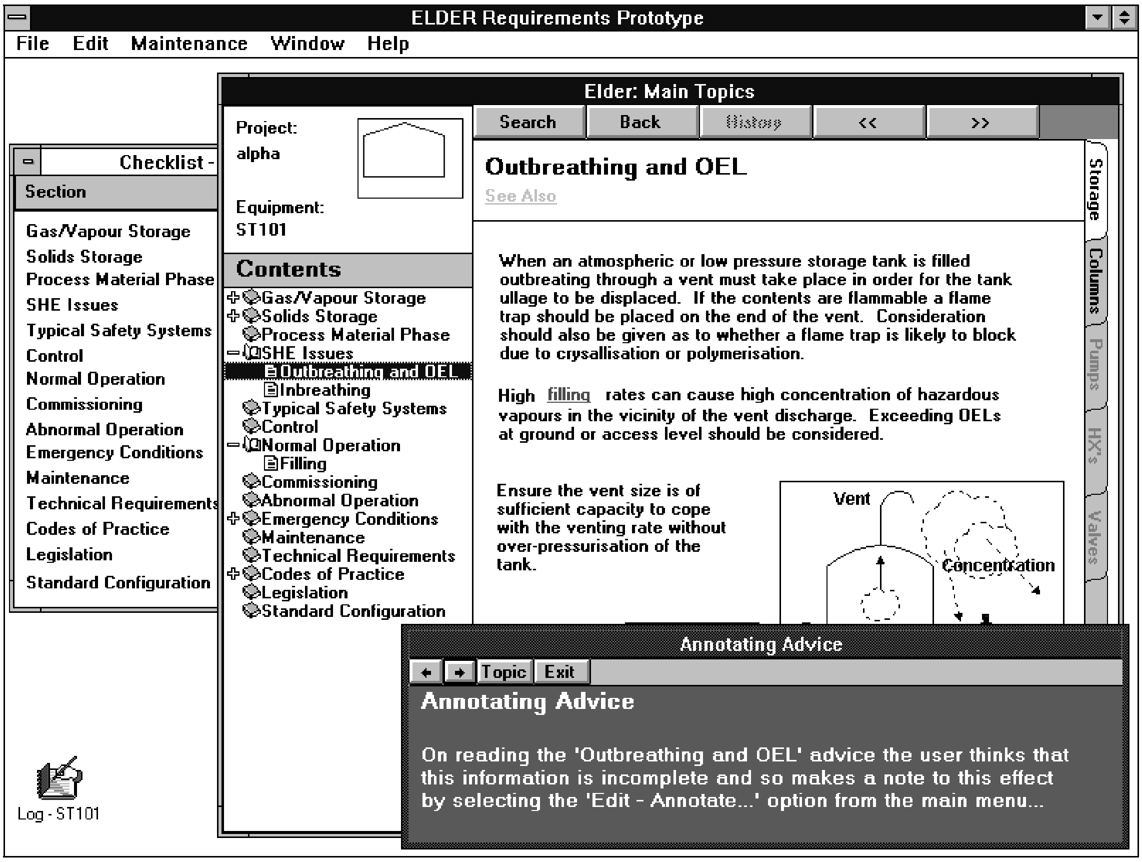

ProtoTour - [c:\..\elder.ani]
File Edit Bookmark Help
Contents Search Back History << >>
ProtoTour
a prototype explanation and visualisation tool
ELDER
Elder Project Information
Elder Design Intent
About ProtoTour
Elder in Action
Elder Style Guide

Figure 3.2 ProtoTour's opening screen presenting the ELDER visual prototype

3.3.3 Narrated Visual Prototype Walkthroughs or ‘Tours’

Any number of pre-recorded walkthrough sequences with accompanying narrative can be played back from within ProtoTour. These walkthroughs mimic an HCI designer’s presentation of a visual prototype. The walkthroughs or ‘tours’ aim to bring to life the visual prototype representation without the need for an experienced operator. Figure 3.3 shows a single frame from an animation sequence, which demonstrates how advice in the ELDER software is annotated by its users. The window shown in the bottom right corner of the figure is the narrative window explaining what the user is seeing in the visual prototype animation. Narrative windows appear after a series of animation frames have shown the manipulation of a particular aspect of the visual prototype interaction. The presence of a narrative window halts the animation until the user decides to progress to the next animated sequence. Alternatively, the user has the option of going back over the last sequence or jumping to a particular sequence.



When an atmospheric or low pressure storage tank is filled outbreating through a vent must take place in order for the tank ullage to be displaced. If the contents are flammable a flame trap should be placed on the end of the vent. Consideration should also be given as to whether a flame trap is likely to block due to crysallisation or polymerisation.

High filling rates can cause high concentration of hazardous vapours in the vicinity of the vent discharge. Exceeding OELs at ground or access level should be considered.

Ensure the vent size is of sufficient capacity to cope with the venting rate without over-pressurisation of the tank.

On reading the 'Outbreathing and OEL' advice the user thinks that this information is incomplete and so makes a note to this effect by selecting the 'Edit - Annotate...' option from the main menu...

Figure 3.3 A single frame from an animation sequence in ELDER

3.3.4 Hypertext Generic Topic Templates

The hypertext generic topic templates themselves are not novel. The hypertext software used in the first implementation of ProtoTour is taken from Microsoft Windows Help. Therefore, the features of the hypertext generic topic templates in ProtoTour are identical to a sophisticated Windows' Help file. Features include:

- free format text which can include hot links[1] to other topics or pop-up sub-topics (e.g. for pop-up definitions);

- a 'See Also' pop-up (see section 3.3.11) uses the features of the above point to list further reading relevant to the current topic;

- pictures may be included in the text and these can include hypertext hotspots[2];

- search terms are associated with each topic which allows a separate search mechanism (see section 3.3.11) to lead the user to relevant topics.

The only difference between a ProtoTour topic and a Help file topic is structural. In order to simplify authoring and maintain consistency, the ProtoTour topics are produced using a more rigidly structured template than typical Help file topics. This also acts as a memory jogger to authors, for example, a 'See Also' and a 'Design Rationale' area is included on every ProtoTour topic page even if it is not used (this would appear to the user as a disabled[3] link).

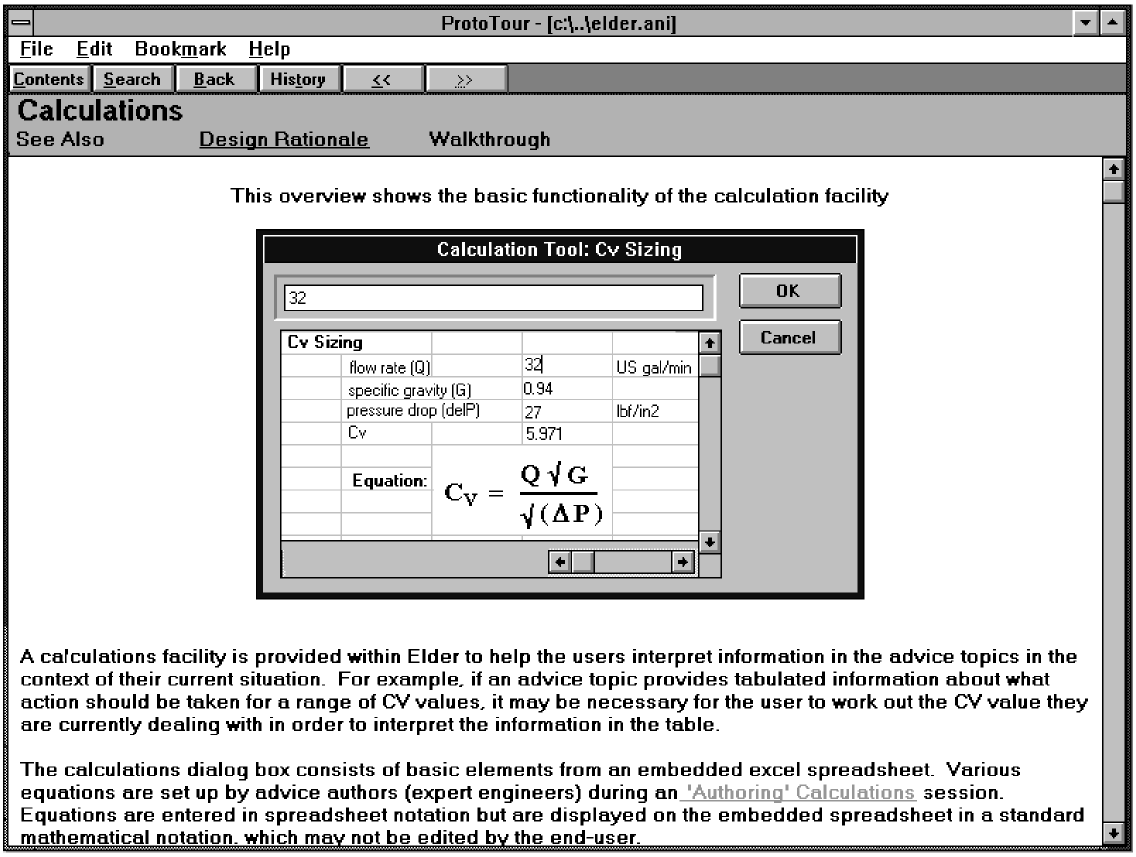
[1] a hot link is a jargon term for a word (usually underlined and in a different colour to the rest of the text) which when clicked upon activates a hypertext event, such as navigating to a different page of hypertext

[2] a hotspot is a jargon term for an area of a picture which when clicked upon activates a hypertext event, such as navigating to a different page of hypertext

[3] a disabled feature in Microsoft Windows jargon refers to a feature which is not currently available to a user but is shown on screen (usually in light grey) to indicate that it exists even though it is unavailable.

3.3.5 Topics Explaining the HCI Design Intent and Functionality Implied By the Visual Prototype

Figure 3.4 illustrates a typical ProtoTour information topic introducing the Calculation facility shown in the ELDER visual prototype. This particular topic comprises a picture of the Calculation Tool dialog box and a description of how it should work. A hot link in the text refers the reader to a topic concerning the ‘Authoring’ of calculations, in other words a topic describing how the users of ELDER will need to change the underlying equations. This is clearly relevant information for a programmer working on the Calculation Tool. In the topic title, there are three more hot links which are common to all ProtoTour information topics. The ‘See Also’ link will refer the reader to other topics which may be related to the Calculation topic. The ‘Design Rationale’ link provides the reader with an explanation of why the Calculation Tool has been designed as it has (and may also explain any unresolved design issues). Finally, the ‘Walkthrough’ hot link will playback a recorded sequence (based on the visual prototype) and an associated narrative, demonstrating how the Calculation Tool has been designed to work in practice.


ProtoTour - [c:\..\elder.ani]
File Edit Bookmark Help
Contents Search Back History << >>
Calculation Tool: Cv Sizing
OK
Cancel

Calculations

See Also Design Rationale Walkthrough

This overview shows the basic functionality of the calculation facility

32

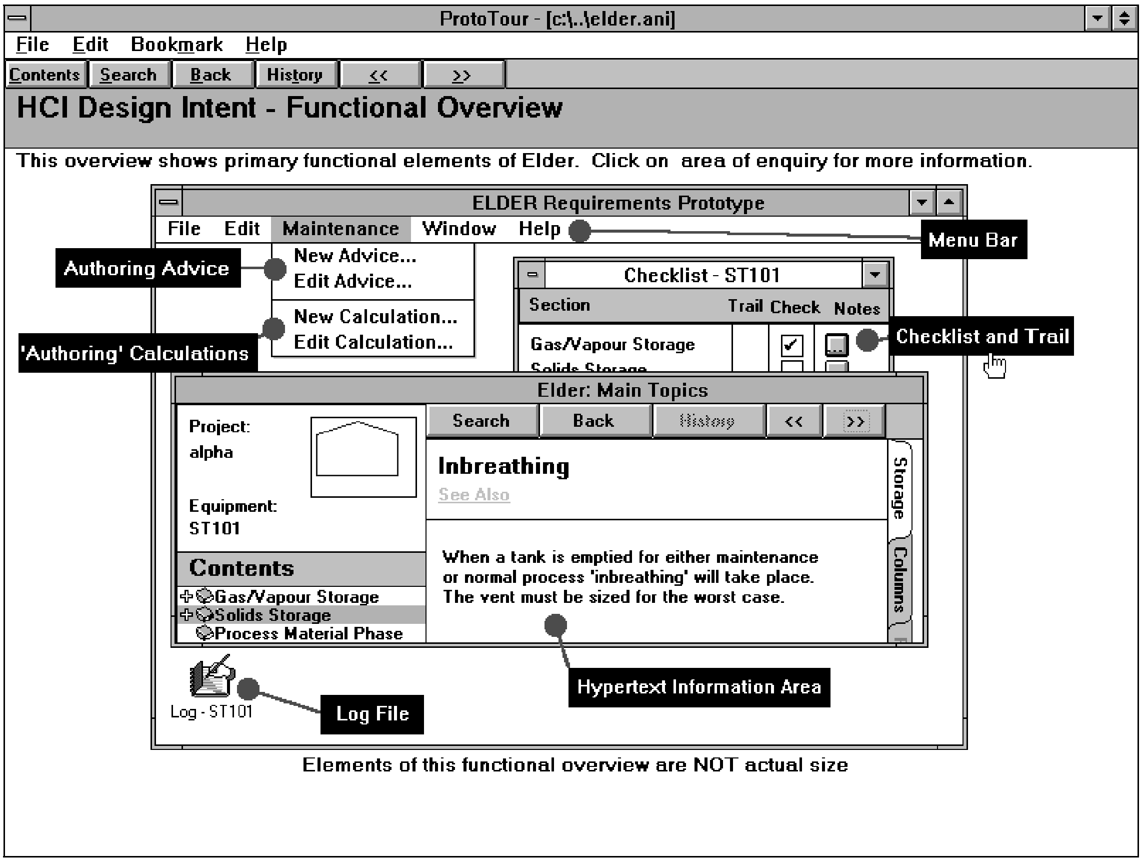
Cv Sizing			
flow rate (Q)		32	US gal/min
specific gravity (G)		0.94	
pressure drop (delP)		27	lbf/in2
Cv		5.971	
Equation:	$C_V = \dfrac{Q\sqrt{G}}{\sqrt{(\Delta P)}}$		

A calculations facility is provided within Elder to help the users interpret information in the advice topics in the context of their current situation. For example, if an advice topic provides tabulated information about what action should be taken for a range of CV values, it may be necessary for the user to work out the CV value they are currently dealing with in order to interpret the information in the table.

The calculations dialog box consists of basic elements from an embedded excel spreadsheet. Various equations are set up by advice authors (expert engineers) during an 'Authoring' Calculations session. Equations are entered in spreadsheet notation but are displayed on the embedded spreadsheet in a standard mathematical notation, which may not be edited by the end-user.

Figure 3.4 Part of a typical ProtoTour information topic

In order to assist the reader's navigation to the relevant aspect of ProtoTour, the author of the ProtoTour representation can construct a picture giving an overview of the visual prototype. This static picture (or series of pictures) can be set up with hypertext hotspots to become a kind of visual contents page for the ProtoTour representation. Figure 3.5 is an example of such a visual contents page. The picture shows the main elements of functionality and interactivity of the ELDER software. Selecting the main 'Hypertext Information Area' hotspot accesses another similar picture, further breaking down the functionality of this complex aspect of ELDER.



HCI Design Intent - Functional Overview

This overview shows primary functional elements of Elder. Click on area of enquiry for more information.

When a tank is emptied for either maintenance or normal process 'inbreathing' will take place. The vent must be sized for the worst case.

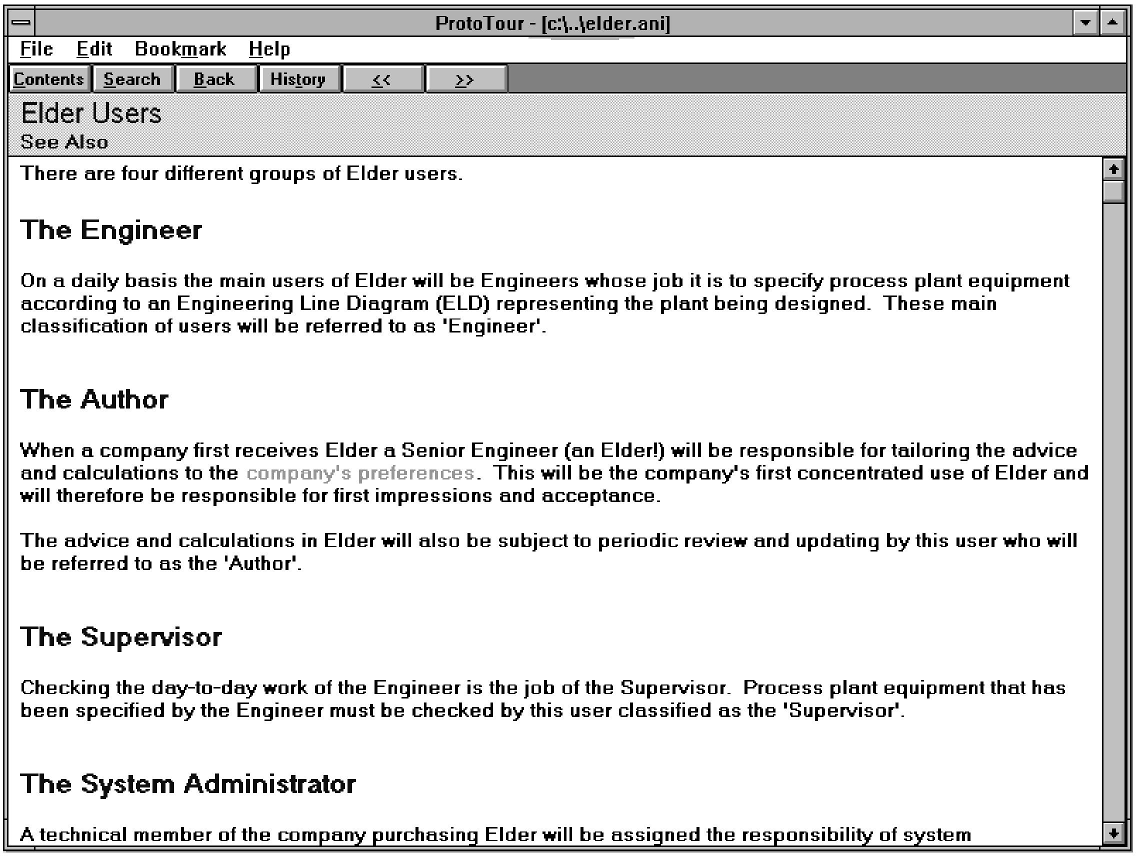
Elements of this functional overview are NOT actual size

Figure 3.5 A ProtoTour topic using a picture to create a visual contents page, giving an overview of functional elements of the proposed software

3.3.6 Topics Providing User and Task Information

ProtoTour topics are also used to convey information about the users of the product illustrated by the visual prototype and their tasks.

Figure 3.6 provides an example of an information topic describing the different users of the proposed ELDER software. Such information about users is sometimes necessary for the interpretation of design rationale information.



Elder Users
See Also

There are four different groups of Elder users.

The Engineer

On a daily basis the main users of Elder will be Engineers whose job it is to specify process plant equipment according to an Engineering Line Diagram (ELD) representing the plant being designed. These main classification of users will be referred to as 'Engineer'.

The Author

When a company first receives Elder a Senior Engineer (an Elder!) will be responsible for tailoring the advice and calculations to the company's preferences. This will be the company's first concentrated use of Elder and will therefore be responsible for first impressions and acceptance.

The advice and calculations in Elder will also be subject to periodic review and updating by this user who will be referred to as the 'Author'.

The Supervisor

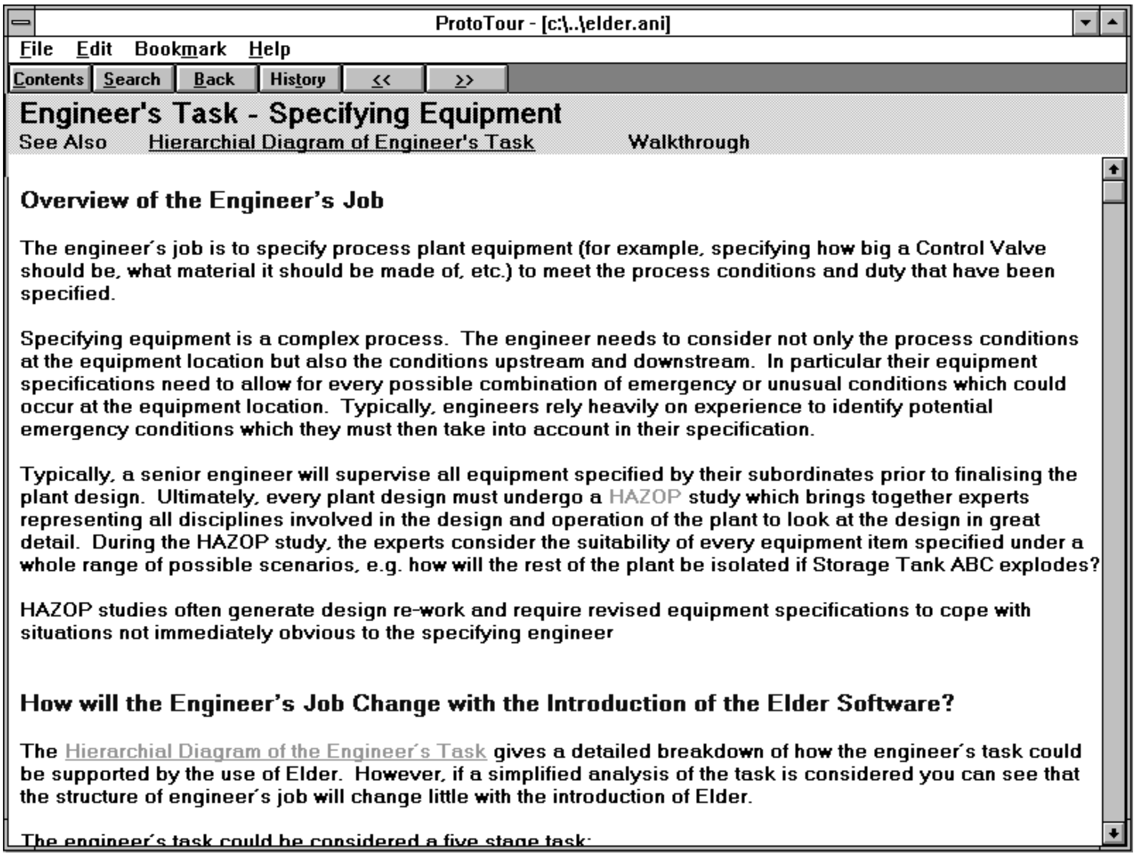
Checking the day-to-day work of the Engineer is the job of the Supervisor. Process plant equipment that has been specified by the Engineer must be checked by this user classified as the 'Supervisor'.

The System Administrator

A technical member of the company purchasing Elder will be assigned the responsibility of system

Figure 3.6 A ProtoTour topic describing the different users of the proposed software

Figure 3.7 gives an example of an information topic describing the main task of the primary user of ELDER. Hot links available within this topic include a 'Hierarchial Diagram of the Engineer's Task'. In this example, the ProtoTour author has included a topic comprising a diagram of their choice (these are discussed further in section 3.3.9).


ProtoTour - [c:\..\elder.ani]
File Edit Bookmark Help
Contents Search Back History << >>

Engineer's Task - Specifying Equipment

See Also Hierarchial Diagram of Engineer's Task Walkthrough

Overview of the Engineer's Job

The engineer's job is to specify process plant equipment (for example, specifying how big a Control Valve should be, what material it should be made of, etc.) to meet the process conditions and duty that have been specified.

Specifying equipment is a complex process. The engineer needs to consider not only the process conditions at the equipment location but also the conditions upstream and downstream. In particular their equipment specifications need to allow for every possible combination of emergency or unusual conditions which could occur at the equipment location. Typically, engineers rely heavily on experience to identify potential emergency conditions which they must then take into account in their specification.

Typically, a senior engineer will supervise all equipment specified by their subordinates prior to finalising the plant design. Ultimately, every plant design must undergo a HAZOP study which brings together experts representing all disciplines involved in the design and operation of the plant to look at the design in great detail. During the HAZOP study, the experts consider the suitability of every equipment item specified under a whole range of possible scenarios, e.g. how will the rest of the plant be isolated if Storage Tank ABC explodes?

HAZOP studies often generate design re-work and require revised equipment specifications to cope with situations not immediately obvious to the specifying engineer

How will the Engineer's Job Change with the Introduction of the Elder Software?

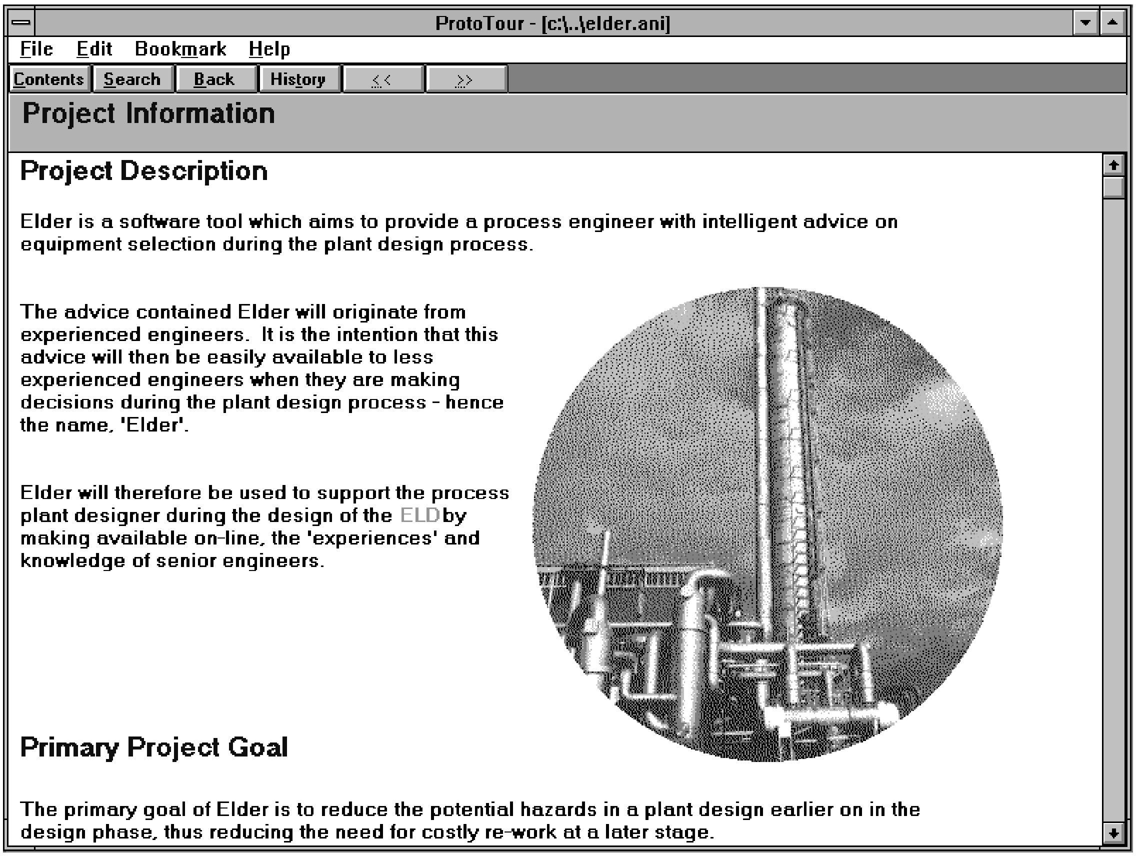
The Hierarchial Diagram of the Engineer's Task gives a detailed breakdown of how the engineer's task could be supported by the use of Elder. However, if a simplified analysis of the task is considered you can see that the structure of engineer's job will change little with the introduction of Elder.

The engineer's task could be considered a five stage task:

Figure 3.7 A ProtoTour topic describing the main task of the primary user of the proposed software

3.3.7 Topics Providing General Information About The Software Project

Statements of the objective, rationale, timescales, and budget are a few examples of the general information about a software project which can be included in a 'Project Information' topic. Figure 3.8 gives an example of some general project information relating to the ELDER project.


ProtoTour - [c:\..\elder.ani]
File Edit Bookmark Help
Contents Search Back History << >>

Project Information

Project Description

Elder is a software tool which aims to provide a process engineer with intelligent advice on equipment selection during the plant design process.

The advice contained Elder will originate from experienced engineers. It is the intention that this advice will then be easily available to less experienced engineers when they are making decisions during the plant design process - hence the name, 'Elder'.

Elder will therefore be used to support the process plant designer during the design of the ELD by making available on-line, the 'experiences' and knowledge of senior engineers.

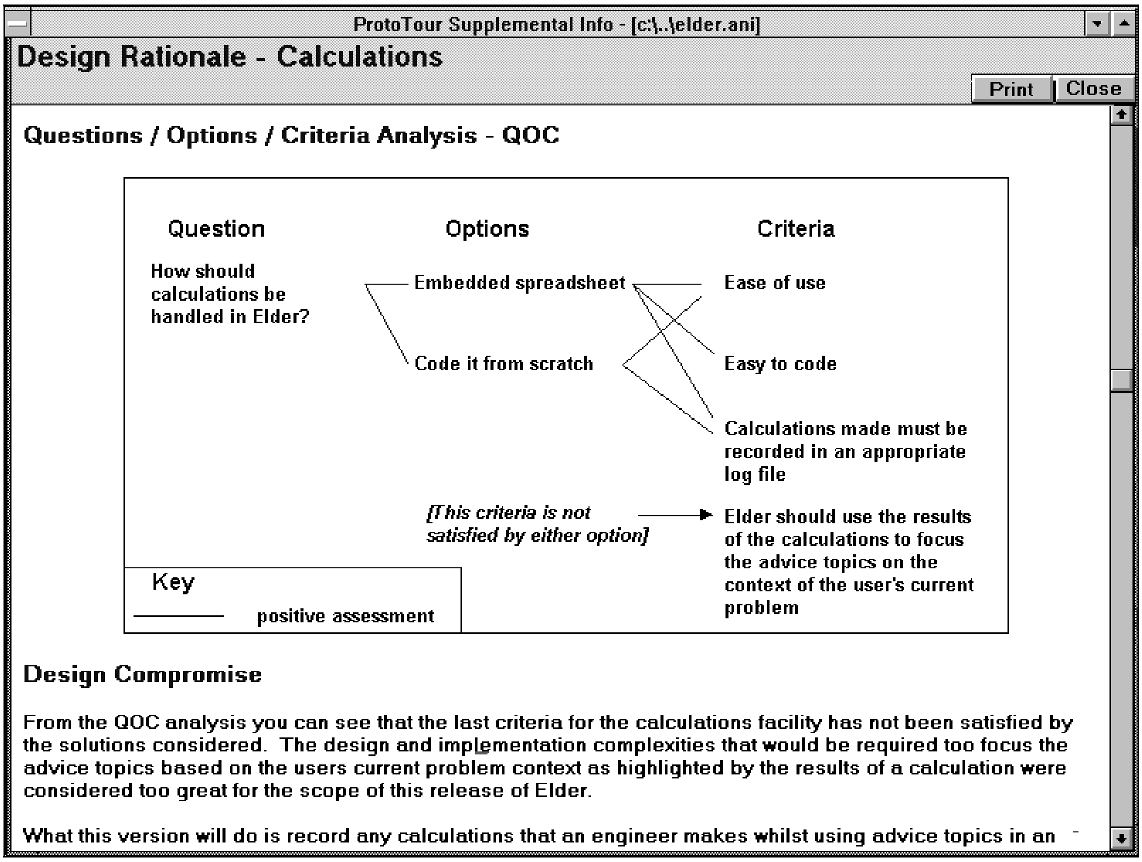
Primary Project Goal

The primary goal of Elder is to reduce the potential hazards in a plant design earlier on in the design phase, thus reducing the need for costly re-work at a later stage.

Figure 3.8 A ProtoTour 'Project Information' topic for the proposed software project

3.3.8 The Representation of Design Rationale

The ProtoTour topic templates include a predefined hot link to a 'Design Rationale' page for every information topic. This emphasises that design rationale is a fundamental aspect of ProtoTour as well as acting as a reminder to authors of a ProtoTour representation. Throughout the ProtoTour ELDER example, design rationale is described in several ways. Firstly, where possible, a simplified version of the QOC representation (see section 1.11.3.4) was used to describe the design considerations made for each interface element. Secondly, written descriptions of design rationale and design consideration were employed. Finally, outstanding design issues or design compromises were recorded. Figure 3.9 shows the part of the design rationale for the Calculation Tool in the form of a QOC representation and a description of design compromises that were made.



ProtoTour Supplemental Info - [c:\..\elder.ani]
Print
Close
Question
Options
Criteria
How should calculations be handled in Elder?
Embedded spreadsheet
Code it from scratch
Ease of use
Easy to code
Calculations made must be recorded in an appropriate log file
[This criteria is not satisfied by either option]
Elder should use the results of the calculations to focus the advice topics on the context of the user's current problem
Key
positive assessment

Design Rationale - Calculations

Questions / Options / Criteria Analysis - QOC

Design Compromise

From the QOC analysis you can see that the last criteria for the calculations facility has not been satisfied by the solutions considered. The design and implementation complexities that would be required too focus the advice topics based on the users current problem context as highlighted by the results of a calculation were considered too great for the scope of this release of Elder.

What this version will do is record any calculations that an engineer makes whilst using advice topics in an

Figure 3.9 Design rationale representation within ProtoTour

3.3.9 Miscellaneous Representations

The ProtoTour author is encouraged to include any representations they see fit to describe the proposed software. ProtoTour topic templates have sufficient flexibility to include various representation, but these are not regarded as part of the ProtoTour concept in the way that design rationale representation is. Figures 3.10 and 3.11 show two kinds of representations which would support explanations within the ProtoTour information. Figure 3.10 is a hierarchial decomposition of the primary task of the main user of ELDER.

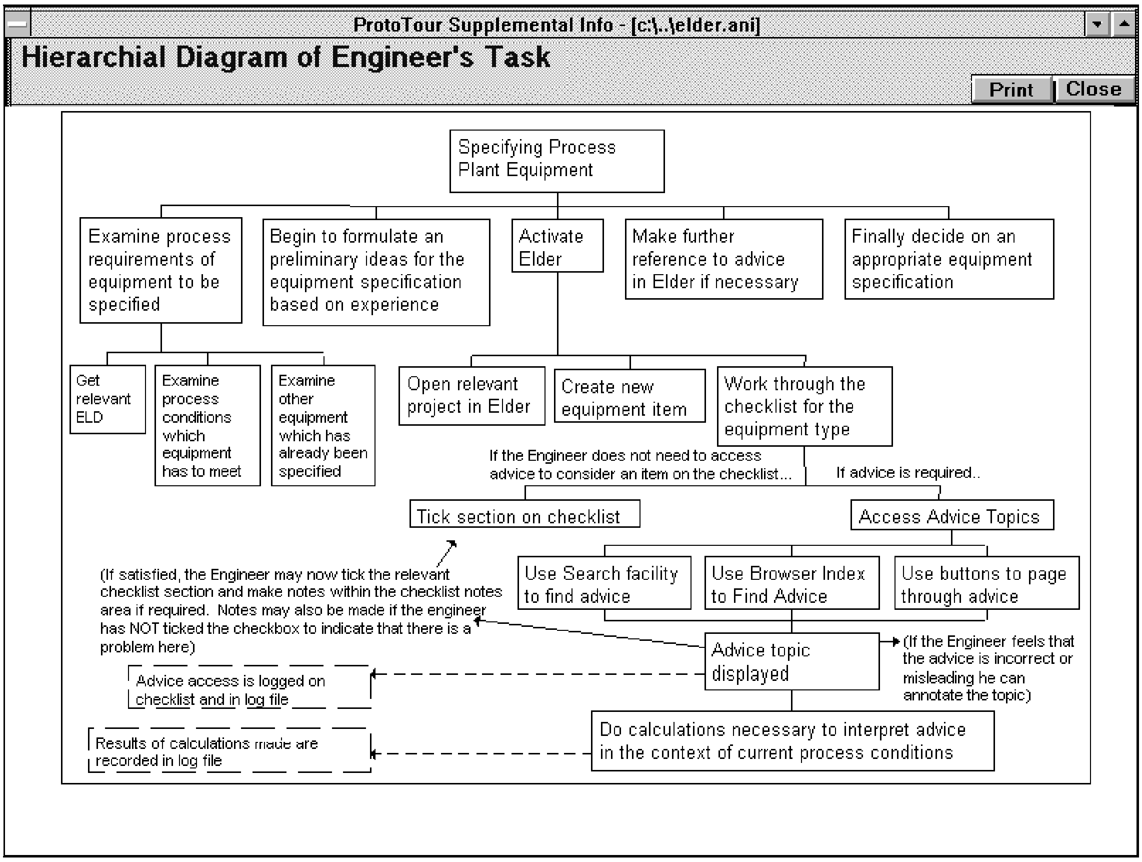
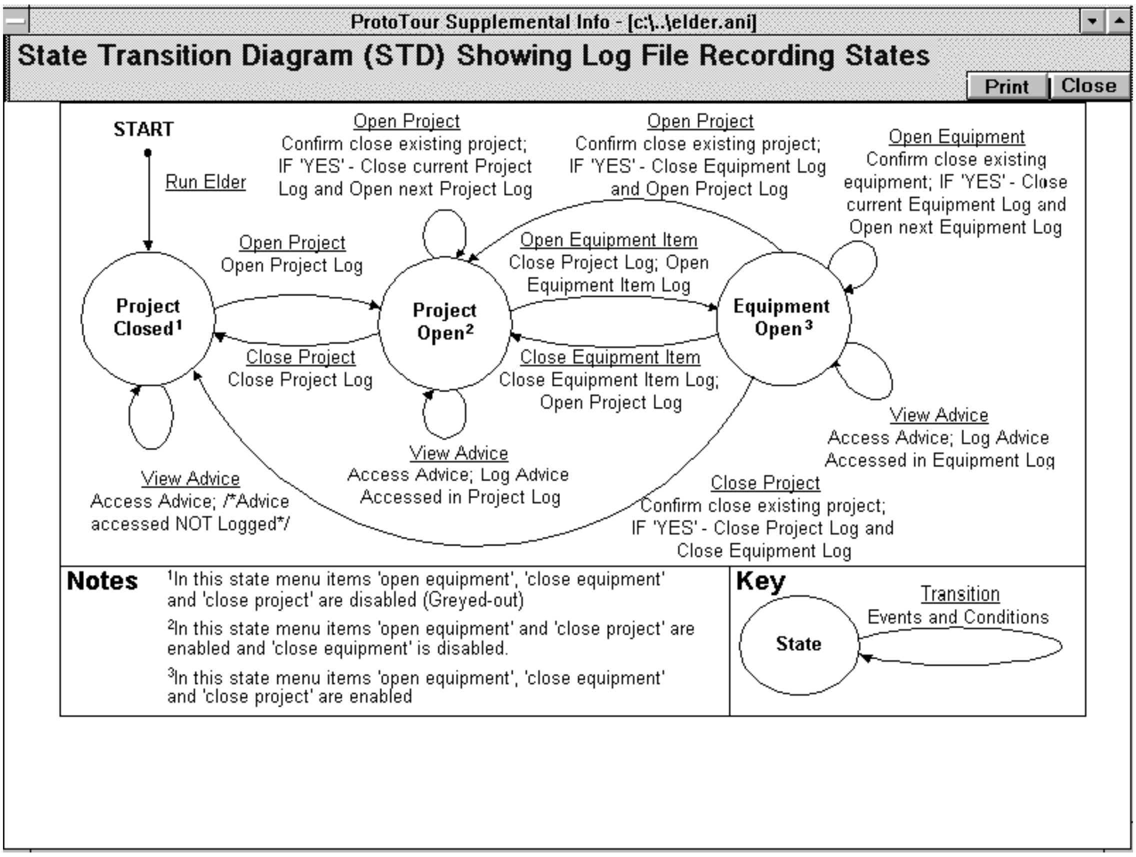

ProtoTour Supplemental Info - [c:\..\elder.ani]
Hierarchial Diagram of Engineer's Task
Print
Close
Specifying Process Plant Equipment
Examine process requirements of equipment to be specified
Begin to formulate an preliminary ideas for the equipment specification based on experience
Activate Elder
Make further reference to advice in Elder if necessary
Finally decide on an appropriate equipment specification
Get relevant ELD
Examine process conditions which equipment has to meet
Examine other equipment which has already been specified
Open relevant project in Elder
Create new equipment item
Work through the checklist for the equipment type
If the Engineer does not need to access advice to consider an item on the checklist...
If advice is required..
Tick section on checklist
Access Advice Topics
(If satisfied, the Engineer may now tick the relevant checklist section and make notes within the checklist notes area if required. Notes may also be made if the engineer has NOT ticked the checkbox to indicate that there is a problem here)
Use Search facility to find advice
Use Browser Index to Find Advice
Use buttons to page through advice
Advice topic displayed
(If the Engineer feels that the advice is incorrect or misleading he can annotate the topic)
Advice access is logged on checklist and in log file
Do calculations necessary to interpret advice in the context of current process conditions
Results of calculations made are recorded in log file

Figure 3.10 Miscellaneous representation - a hierarchial decomposition of the primary task of the main user of the proposed software

Figure 3.11 is a State-Transition Diagram (STD) which accurately describes the status of Log Files in ELDER. This diagram is typical of the type of design work which is often duplicated by the programmer, because the visual prototype alone does not convey the level of design work that has already gone in to various aspects of the software.


ProtoTour Supplemental Info - [c:\..\elder.ani]
Print
Close
START
Run Elder
Project Closed¹
Project Open²
Equipment Open³
Open Project
Open Project Log
Close Project
Close Project Log
Open Project
Confirm close existing project; IF 'YES' - Close current Project Log and Open next Project Log
Open Project
Confirm close existing project; IF 'YES' - Close Equipment Log and Open Project Log
Open Equipment
Confirm close existing equipment; IF 'YES' - Close current Equipment Log and Open next Equipment Log
Open Equipment Item
Close Project Log; Open Equipment Item Log
Close Equipment Item
Close Equipment Item Log; Open Project Log
View Advice
Access Advice; Log Advice Accessed in Equipment Log
Close Project
Confirm close existing project; IF 'YES' - Close Project Log and Close Equipment Log
View Advice
Access Advice; Log Advice Accessed in Project Log
View Advice
Access Advice; /*Advice accessed NOT Logged*/
Key
State
Transition
Events and Conditions

State Transition Diagram (STD) Showing Log File Recording States

Notes

[1] In this state menu items 'open equipment', 'close equipment' and 'close project' are disabled (Greyed-out)

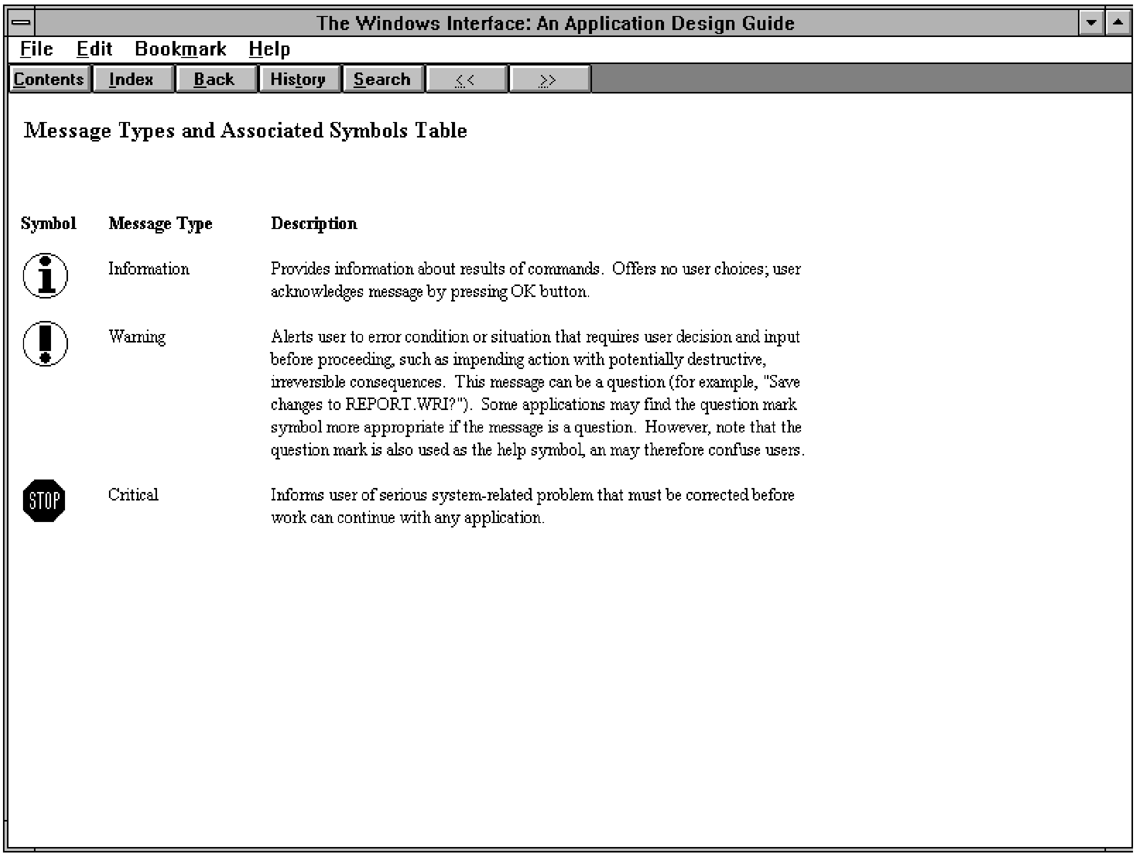
[2] In this state menu items 'open equipment' and 'close project' are enabled and 'close equipment' is disabled.

[3] In this state menu items 'open equipment', 'close equipment' and 'close project' are enabled

Figure 3.11 Miscellaneous representation - a State-Transition Diagram relating to an aspect of the proposed software

3.3.10 Style Guide

ProtoTour has been designed to facilitate access to an on-line style guide. The style guide linked to the ProtoTour representation of the ELDER visual prototype is Microsoft's Windows Interface Application Design Guide. Figure 3.12 shows a sample page of information from the style guide.



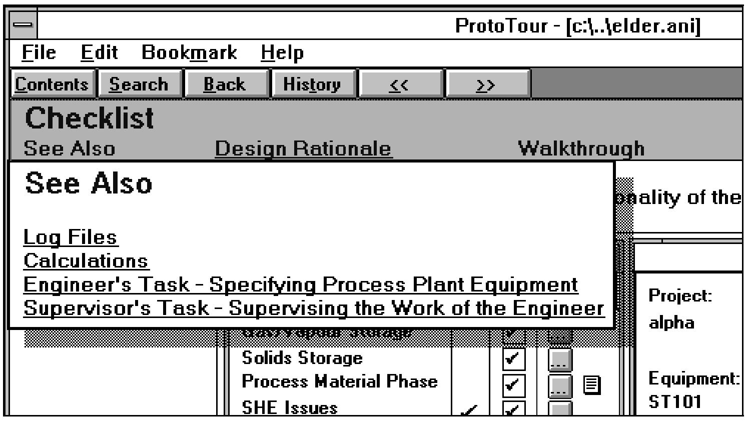
Message Types and Associated Symbols Table

Symbol	Message Type	Description
ⓘ	Information	Provides information about results of commands. Offers no user choices; user acknowledges message by pressing OK button.
❗	Warning	Alerts user to error condition or situation that requires user decision and input before proceeding, such as impending action with potentially destructive, irreversible consequences. This message can be a question (for example, "Save changes to REPORT.WRI?"). Some applications may find the question mark symbol more appropriate if the message is a question. However, note that the question mark is also used as the help symbol, an may therefore confuse users.
STOP	Critical	Informs user of serious system-related problem that must be corrected before work can continue with any application.

Figure 3.12 A sample page from a style guide linked to the ProtoTour representation

3.3.11 Cross-Referencing and Searching

A ‘See Also’ pop-up window is part of every information topic which allows the author of the ProtoTour representation to suggest other relevant, alternative topics to the reader. Figure 3.13 shows an example of a ‘See Also’ pop-up window suggesting alternative topics to the reader of the Checklist topic in the ELDER prototype.



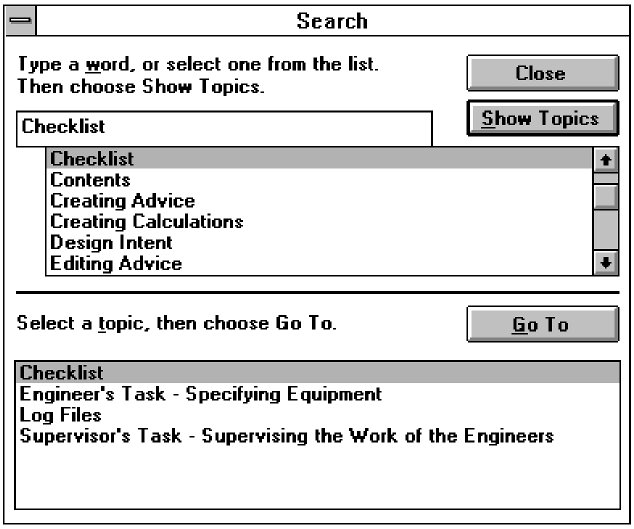
Checklist

See Also

Log Files
Calculations
Engineer's Task - Specifying Process Plant Equipment
Supervisor's Task - Supervising the Work of the Engineer

Figure 3.13 An example ‘See Also’ pop-up window

Search terms associated with each ProtoTour topic provide the user with a non-hypertext style index to the information in the ProtoTour representation. Figure 3.14 shows an example of this search mechanism, showing the four topics in the ProtoTour representation of ELDER which contain the search term ‘Checklist’.



Type a word, or select one from the list.
Then choose Show Topics.

Checklist

Checklist
Contents
Creating Advice
Creating Calculations
Design Intent
Editing Advice

Select a topic, then choose Go To.

Checklist
Engineer's Task - Specifying Equipment
Log Files
Supervisor's Task - Supervising the Work of the Engineers

Figure 3.14 The ProtoTour search mechanism

3.4 Implementation

This section describes the implementation of the first prototype of ProtoTour.

3.4.1 Implementation Overview

An overview of the main functional aspects of ProtoTour, depicted in figure 3.15, is described below.

The heart of ProtoTour is the hypertext engine; in this case the Window's Help application (known as winhelp). Pre-prepared and compiled ProtoTour hypertext help files are run within winhelp.

Before running winhelp, a separate loader program (written in Visual Basic) changes the Windows colour scheme to a specific ProtoTour colour scheme (this enables the user to distinguish ProtoTour explanation from the visual prototype animation sequences). The loader program remains running in the background, ready to restore the original colour scheme when ProtoTour exits.

When winhelp is run, the appropriate ProtoTour hypertext file is also loaded. The user may now interrogate the information in the hypertext file as required.

Only when the user selects a walkthrough link from the hypertext file are the programs necessary to set up and play the animation sequences loaded. The first program to load and run is another Visual Basic loader program. This program first tidies the display by iconising the winhelp application (this is necessary because it is set to be the topmost window when ProtoTour is running) and the program manager. Secondly, a splash screen announces that animation sequences are loading, followed by a set of simple instructions for navigating around the animations. Thirdly, the program loads the DemoQuick application (the application which allows sequences from a visual prototype to be recorded and played back) and the relevant animation file. The loader program remains running in the background ready to restore winhelp after the animation sequence has finished and DemoQuick has been exited. The loader program then exits and returns control to the winhelp application.

There was one final complication to the ProtoTour implementation, which is not shown in figure 3.15 for the sake of simplicity. The style guide which was linked to ProtoTour was a Microsoft Multimedia Viewer file. Thus a further loader program was written in Visual Basic which activated the Multimedia Viewer application and loaded the style guide. As with the other programs, when the style guide was closed, it returned control to the winhelp application.

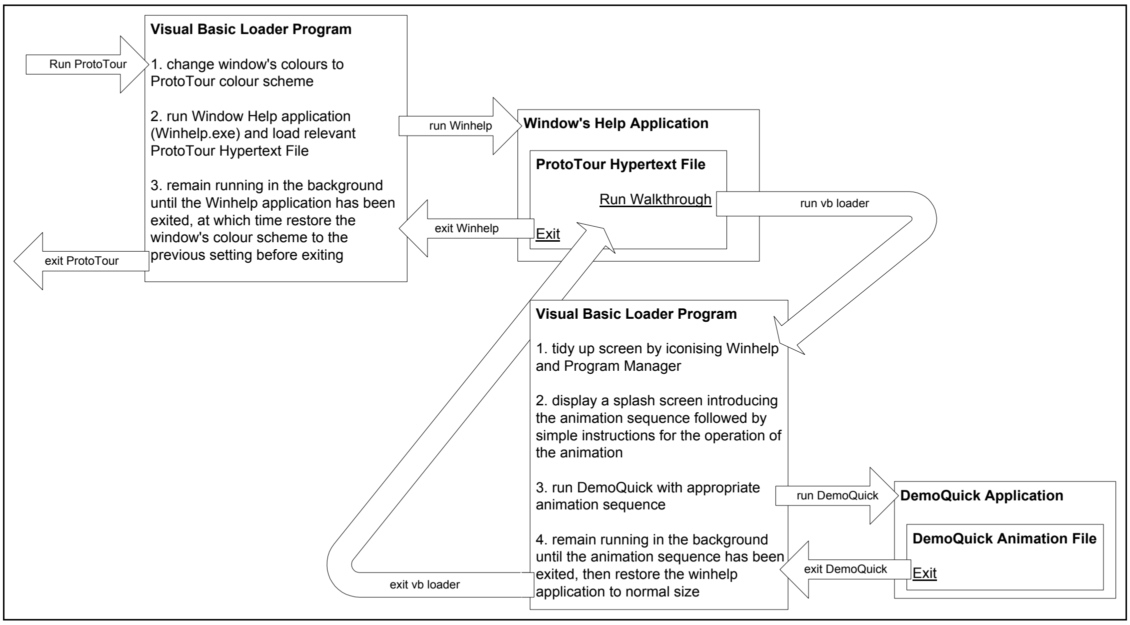

Run ProtoTour
Visual Basic Loader Program
1. change window's colours to ProtoTour colour scheme
2. run Window Help application (Winhelp.exe) and load relevant ProtoTour Hypertext File
3. remain running in the background until the Winhelp application has been exited, at which time restore the window's colour scheme to the previous setting before exiting
exit ProtoTour
run Winhelp
exit Winhelp
Window's Help Application
ProtoTour Hypertext File
Run Walkthrough
Exit
run vb loader
Visual Basic Loader Program
1. tidy up screen by iconising Winhelp and Program Manager
2. display a splash screen introducing the animation sequence followed by simple instructions for the operation of the animation
3. run DemoQuick with appropriate animation sequence
4. remain running in the background until the animation sequence has been exited, then restore the winhelp application to normal size
exit vb loader
run DemoQuick
exit DemoQuick
DemoQuick Application
DemoQuick Animation File
Exit

Figure 3.15 Overview of the ProtoTour implementation

The implementation of ProtoTour outlined in this section is more complicated than it would have been if state-of-the-art tools had been available. For example, a version of DemoQuick which integrates animation sequences with Windows Help is available (although expensive) but creative use of Microsoft Visual Basic enabled a loader program to be written which crudely linked the two applications, bypassing the need for the top-of-the-range DemoQuick version. A flexible ProtoTour application would consist of a single application with facilities for hypertext as well as animation sequences.

3.4.2 Preparation of ProtoTour Hypertext Files

ProtoTour hypertext files are exactly the same as Windows Help files. However, ProtoTour hypertext files make use of a greater proportion of the technical features of Windows Help than is usually the case with Windows Help files.

Figure 3.16 gives an overview of the various applications (and intermediate files) that are used to create a single ProtoTour hypertext file.

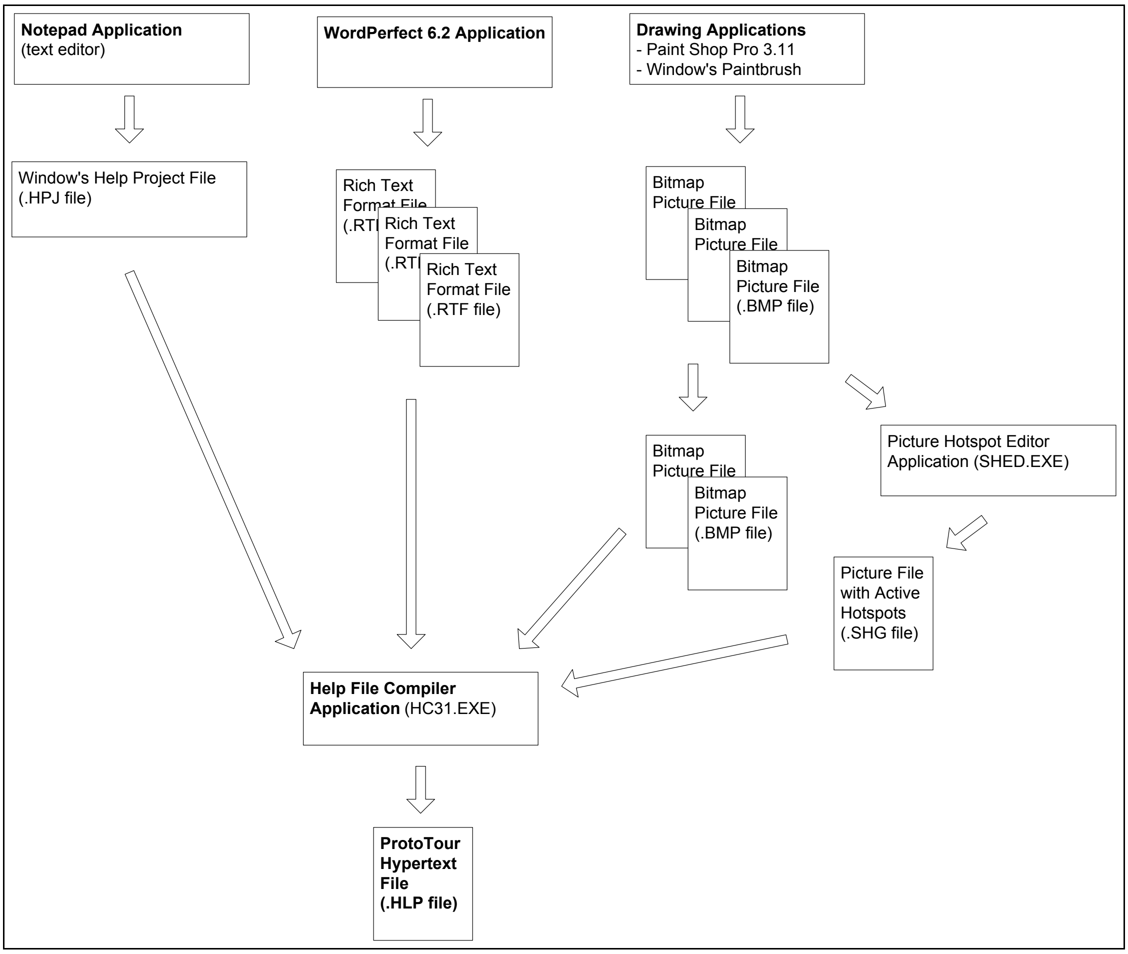
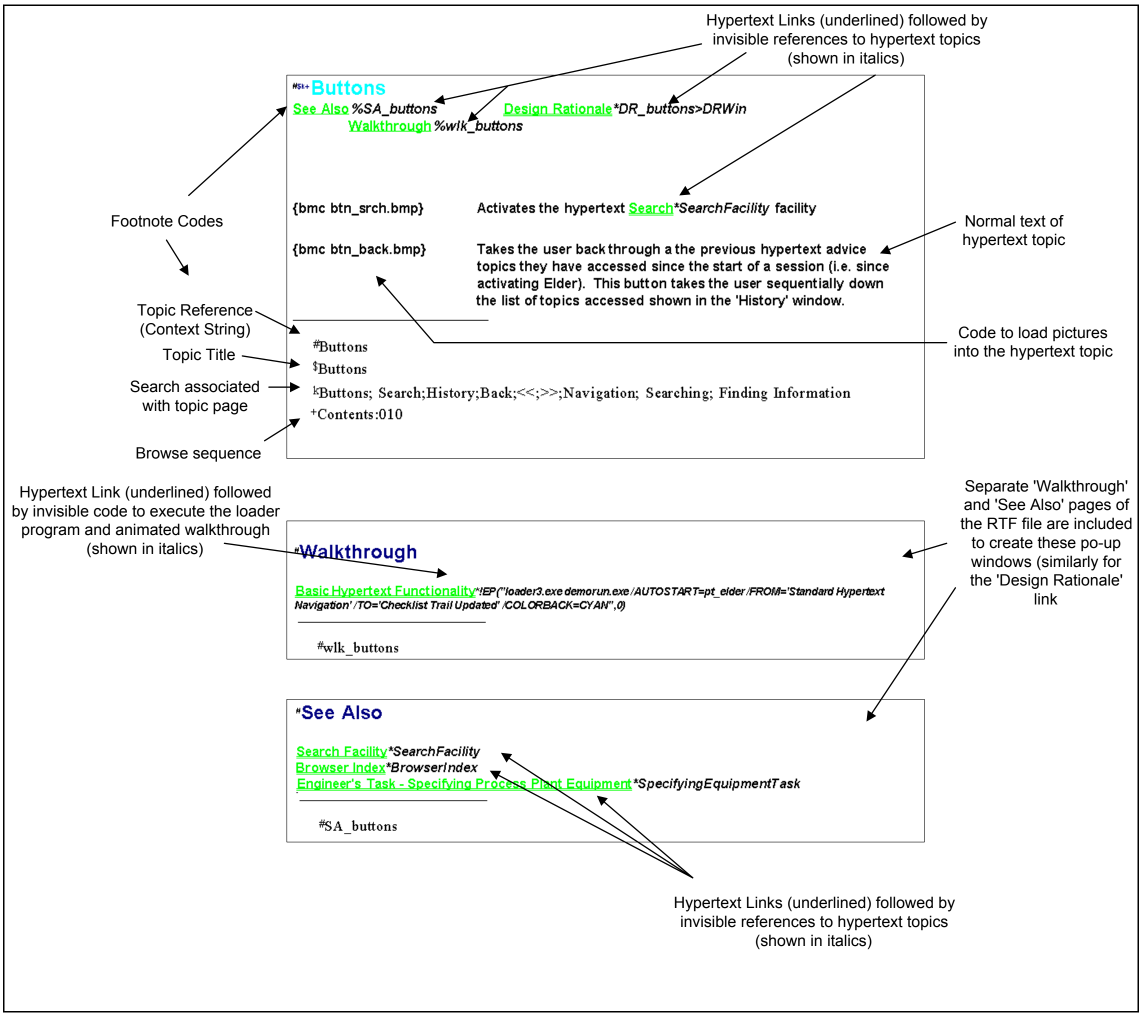

Notepad Application
(text editor)
WordPerfect 6.2 Application
Drawing Applications
- Paint Shop Pro 3.11
- Window's Paintbrush
Window's Help Project File
(.HPJ file)
Rich Text Format File (.RTF file)
Bitmap Picture File (.BMP file)
Picture Hotspot Editor Application (SHED.EXE)
Picture File with Active Hotspots (.SHG file)
Help File Compiler Application (HC31.EXE)
ProtoTour Hypertext File (.HLP file)

Figure 3.16 An overview of the applications and intermediate files which are used in the creation of a ProtoTour hypertext file

The starting point in the process is the production of Rich Text Format (.RTF) files, written in WordPerfect (Microsoft Word would probably have been more appropriate as it provides better support for writing help files; unfortunately, it was not available to the researcher). They contain all the information that will appear in the hypertext and references to any diagrams that will reside with the text. All other hypertext features are encoded in the file in a format prescribed by the Windows Help Application. An example of a ProtoTour hypertext topic RTF file is shown in figure 3.17.



#$k+Buttons

See Also%SA_buttons Design Rationale*DR_buttons>DRWin

Walkthrough%wlk_buttons

{bmc btn_srch.bmp} Activates the hypertext Search*SearchFacility facility

{bmc btn_back.bmp} Takes the user back through a the previous hypertext advice topics they have accessed since the start of a session (i.e. since activating Elder). This button takes the user sequentially down the list of topics accessed shown in the 'History' window.

#Buttons

$Buttons

kButtons; Search;History;Back;<<;>>;Navigation; Searching; Finding Information

+Contents:010

#Walkthrough

Basic Hypertext Functionality*!EP("loader3.exe demorun.exe /AUTOSTART=pt_elder /FROM='Standard Hypertext Navigation' /TO='Checklist Trail Updated' /COLORBACK=CYAN",0)

#wlk_buttons

#See Also

Search Facility*SearchFacility
Browser Index*BrowserIndex
Engineer's Task - Specifying Process Plant Equipment*SpecifyingEquipmentTask

#SA_buttons

Hypertext Links (underlined) followed by invisible references to hypertext topics (shown in italics)
Footnote Codes
Normal text of hypertext topic
Topic Reference (Context String)
Topic Title
Search associated with topic page
Browse sequence
Code to load pictures into the hypertext topic
Hypertext Link (underlined) followed by invisible code to execute the loader program and animated walkthrough (shown in italics)
Separate 'Walkthrough' and 'See Also' pages of the RTF file are included to create these po-up windows (similarly for the 'Design Rationale' link
Hypertext Links (underlined) followed by invisible references to hypertext topics (shown in italics)

Figure 3.17 An example of a ProtoTour hypertext topic RTF file

The RTF file example in figure 3.17 shows that document footnotes are used to encode various information associated with the hypertext topic, ranging from the name and internal reference of the topic to search terms associated with it. The actual text of the topic is included in the file approximately as it should appear in the final hypertext file. Code to load pictures is included in braces. Hypertext links are coded by underlining (or double underlining) the link word and following this with a hidden text reference to the appropriate hypertext destination. Alternatively, some hypertext links execute a Visual Basic loader program which runs a DemoQuick animation sequence. Pop-up windows (like ‘See Also’) and secondary windows (like ‘Design Rationale’) appear on separate pages of the RTF file.

With reference to figure 3.16, drawing packages are used to create pictures to be included in the hypertext file. Paint Shop Pro version 3.11 is of particular use for capturing images from the visual prototype. Paintbrush can subsequently be used to touch-up such images. Other drawing packages (such as Visio 3.0) were used to create diagrams (for example, State-Transition Diagrams), which were exported as bitmaps to be incorporated in the hypertext file. Selected picture files (bitmaps or BMP files) were manipulated with the 'Hotspot Editor' to designate parts of the picture as hypertext hotspots, which could then be linked to other hypertext features.

Before compiling the hypertext file, the final preparatory step is the creation of a project file (or HPJ file) in a text editor (e.g. notepad). This defines the contents of the compiled hypertext file in terms of the collection of RTF files it is comprised of, the naming of the hypertext windows, the font to be used in the hypertext and other parameters.

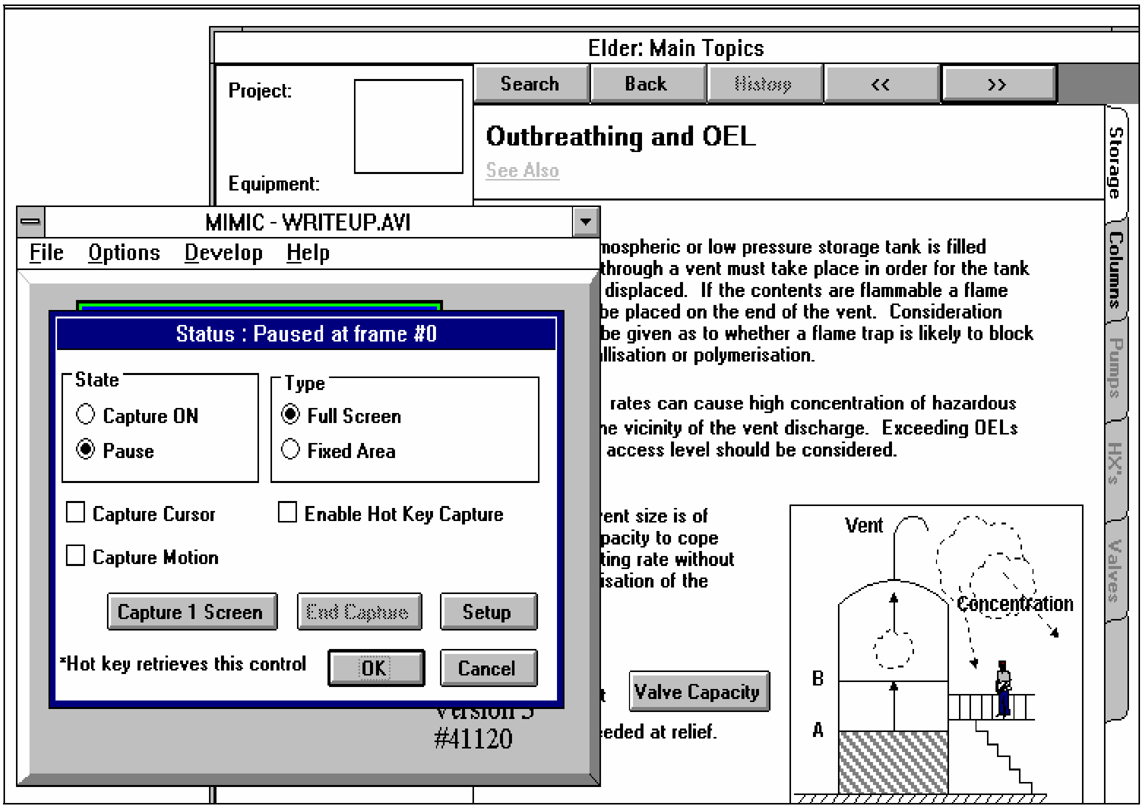
Finally, the Help File Compiler (HC31) was used to compile all RTF files and pictures into the single ProtoTour hypertext file.

As with the overall implementation of ProtoTour, hypertext files can be produced via a less torturous route using advanced tools like RoboHelp by Blue Sky. Such tools enable hypertext to be produced without the need for the author to be directly involved in the complicated encoding formats of the RTF files shown in figure 3.17 or the complicated compiling procedure. Although these tools existed at the time when the ProtoTour prototype was constructed, they were designed to work with Microsoft Word and were quite expensive in their own right.

3.4.3 Preparation of DemoQuick Animated Walkthroughs

Four tools from the DemoQuick suite (produced by The AMT Corporation) were utilised to produce animated walkthroughs of the visual prototype for ProtoTour. The 'MIMIC' tool was used to record sequences of action from the visual prototype in audio-visual interleaved (AVI) files. The 'CLIPquick AVI Editor' enabled animation frames in the AVI files to be edited. 'DemoQuick' was used to add narration pop-up windows to the AVI files and re-create the mouse events from the visual prototype animation. 'DemoQuick' was also used to bring together several AVI files and to compile the resultant sequences into a coherent animated walkthrough. Finally, 'DemoRun' was used to run the completed walkthroughs.

Figure 3.18 shows the MIMIC tool being used to record a sequence of events from the ELDER visual prototype.



Status : Paused at frame #0

State
○ Capture ON
● Pause

Type
● Full Screen
○ Fixed Area

□ Capture Cursor
□ Enable Hot Key Capture
□ Capture Motion

*Hot key retrieves this control

nospheric or low pressure storage tank is filled through a vent must take place in order for the tank displaced. If the contents are flammable a flame be placed on the end of the vent. Consideration be given as to whether a flame trap is likely to block llisation or polymerisation.

rates can cause high concentration of hazardous he vicinity of the vent discharge. Exceeding OELs access level should be considered.

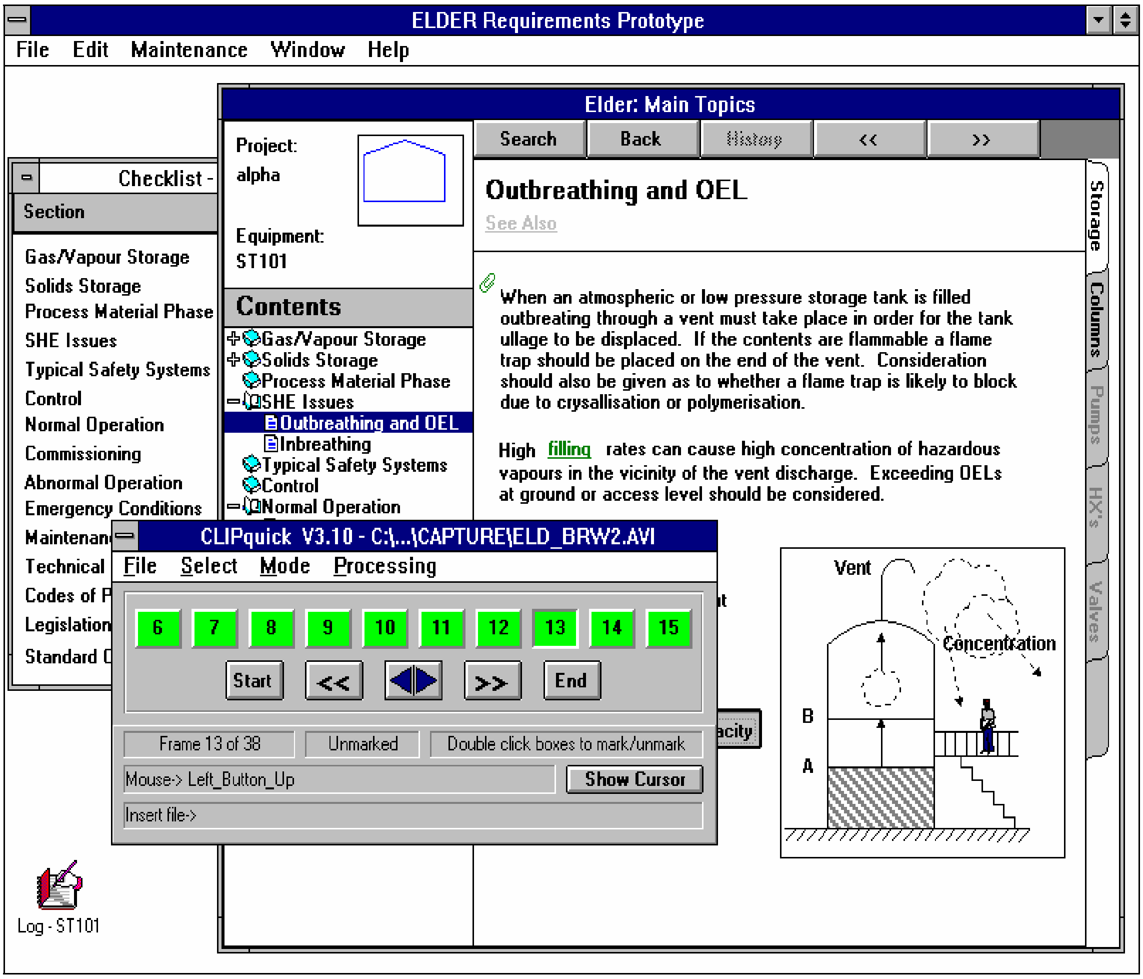
ent size is of pacity to cope ting rate without isation of the

t

eded at relief.

Figure 3.18 MIMIC tool recording a sequence of events

In figure 3.19 the ClipQuick tool is shown editing a frame in a sequence from an AVI file illustrating the ELDER visual prototype.



Project: alpha

Equipment: ST101

Outbreathing and OEL

When an atmospheric or low pressure storage tank is filled outbreating through a vent must take place in order for the tank ullage to be displaced. If the contents are flammable a flame trap should be placed on the end of the vent. Consideration should also be given as to whether a flame trap is likely to block due to crysallisation or polymerisation.

High filling rates can cause high concentration of hazardous vapours in the vicinity of the vent discharge. Exceeding OELs at ground or access level should be considered.

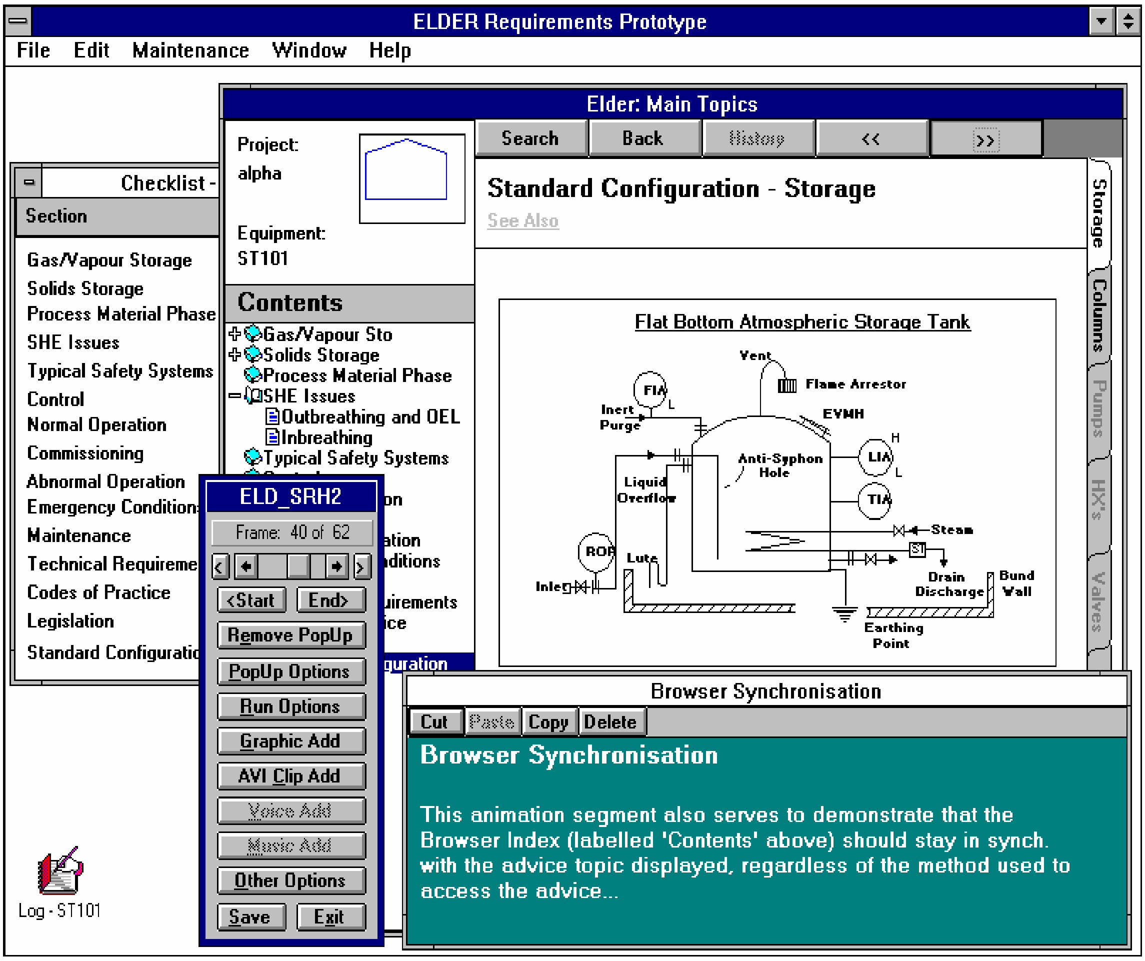
Frame 13 of 38 | Unmarked | Double click boxes to mark/unmark

Mouse-> Left_Button_Up

Insert file->

Figure 3.19 ClipQuick tool editing a frame in a sequence from an AVI file

The DemoRun tool shown in figure 3.20 is being used to add a narrative pop-up description of the Browser Synchronisation in the ELDER visual prototype.

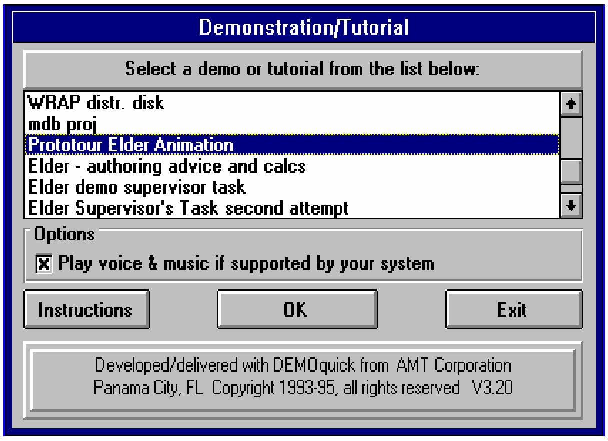


Browser Synchronisation

This animation segment also serves to demonstrate that the Browser Index (labelled 'Contents' above) should stay in synch. with the advice topic displayed, regardless of the method used to access the advice...

Figure 3.20 DemoQuick tool adding a narrative pop-up

Finally, the DemoRun tool is shown in figure 3.21 showing a list of completed animated walkthroughs which can be run.



Select a demo or tutorial from the list below:

WRAP distr. disk
mdb proj
Prototour Elder Animation
Elder - authoring advice and calcs
Elder demo supervisor task
Elder Supervisor's Task second attempt

Options
☒ Play voice & music if supported by your system

Developed/delivered with DEMOquick from AMT Corporation
Panama City, FL Copyright 1993-95, all rights reserved V3.20

Figure 3.21 The DemoRun tool

3.4.4 Implementation Constraints

As in most software developments, several technical constraints were experienced during the implementation. The main ones are described below.

1. Changing the colour of the user's window colour scheme was the only viable method of changing the colour of the ProtoTour windows. This was necessary so that the Windows Help application and the DemoQuick narration pop-ups took on a consistent appearance, which was visually distinct from the windows of the visual prototype.

2. A 16 colour display with a resolution of 800 by 600 pixels was necessary to reduce the memory requirements of the animation sequences.

3. Limitations were experienced on the colour schemes which were controllable within the Windows Help application, with the colour of hypertext links being particularly troublesome.

4. WordPerfect 6.2 does not facilitate the complete range of features required to utilise the full features of the Windows Help application. For example, it is not possible to specify a screen-based font for an RTF file; this is ironic given that Help Files are usually read on-screen (Microsoft Word does not suffer from this constraint).

5. WordPerfect 6.2 provides very little documentation for the construction of Windows Help files and no more is available from the technical support team. in contrast, a great deal of technical information regarding Microsoft products is available on the Microsoft Developers Network CD-ROM.

6. The DemoQuick suite of tools suffered stability problems and bugs, some of which were worked-around by delving into more technical elements of the tools than their authors intended. For example, rogue mouse events were eliminated by breaking in to the mouse event data file and manually changing the data.

It seems likely that Microsoft Word would have been a better tool for creating the hypertext files. Finally, it is worth noting that the Microsoft Windows Help application (used at the heart of ProtoTour) proved to be extremely robust and programmable. It imposed very few constraints on the development of ProtoTour which, technically, took it to its limits.

3.4.5 Implementation of a ProtoTour Representation Took a Long Time

Although not a research result, it is an important observation that it took a long time to produce a ProtoTour representation of the ELDER visual prototype. Part of the construction involved dealing with implementation issues for the first time; for example, changing the windows colours when ProtoTour was running. However, producing the information to support an explanation of the ELDER visual prototype and the accompanying animation sequences was the most time consuming part. This time scale was incurred despite the fact that the researcher (and ProtoTour developer) was the HCI designer who designed and constructed the ELDER visual prototype and so had a deep understanding of the conceptual model it reflected and design rationale.

Using better tools to produce the hypertext files and animated walkthroughs would undoubtedly speed up the production of a ProtoTour representation of a visual prototype. However, producing such a detailed and thorough explanation is unlikely to ever be as rapid as producing the visual prototypes to which they relate.

3.5 Formative Evaluation

During the design of ProtoTour, formative evaluation was undertaken and caused some changes to the eventual implementation. The main evaluation, which was semi-formal, was carried out by demonstrating ProtoTour to the evaluator (an HCI practitioner with both academic and commercial experience), who subsequently investigated its features hands-on and unguided. Informal follow-up evaluations were conducted with another HCI practitioner and several software developers to gain more insight into the problems discovered during the main evaluation, and to ascertain whether corrective measures were successful.

The main findings from the evaluation were as follows.

1. The evaluator was surprised at certain things which were felt to be necessary parts of the ProtoTour explanation. For example, the detailed specification of standard Microsoft Windows controls.

2. The evaluator experienced some conceptual difficulties due to the multi-modality of ProtoTour. In particular, there was confusion over which parts of the software were ProtoTour and which parts were the ELDER visual prototype. This problem proved particularly confusing during the playback of the animated walkthroughs, as the narrative window appeared to be just another window in the prototype application (which filled the screen).

3. The evaluator found it difficult to ascertain which elements of the pictures in the ProtoTour explanation were Hotspots.

Changes to the implementation of ProtoTour in response to the findings from the evaluation were made as follows:

1. No change to ProtoTour. Explanations at the level shown in the ELDER example ProtoTour representation were deemed appropriate. The level of detail to which the author elects to go is left to their own discretion and understanding of the skills and abilities present in their implementation team.

2. Further assessment of the confusion due to multi-modality with other evaluators showed this to be an important problem. This primary finding indicated that ProtoTour had to look distinctly different from the visual prototype it was explaining. A combination of measures was taken to address this:

    - the colour of the ProtoTour window surround, text, headings and paper were changed to look different from the same features of the visual prototype shown in the explanation (this proved a technically difficult requirement to satisfy);

    - an introductory screen presented the concept of ProtoTour as a film projector on which the ELDER visual prototype was ‘Now Showing’ (see figure 3.2);

- the title bar of the ProtoTour application was changed to present the illusion that the visual prototype of the proposed software was merely a file loaded into the ProtoTour tool, i.e. ‘ProtoTour - [c:\..\elder.ani]’;

- an introductory ‘splash’ screen and operating instructions were added to precede the loading of the animation sequences (a necessarily disjointed activity, while the Windows Help application is iconised and the DemoQuick application is loaded).

3. Hotspots were made more obvious in pictures. This was partially achieved by using a consistent labelling format for pictures and making the labels the same colour as other hypertext links in ProtoTour.

With limited further evaluation, these changes ironed-out the main usability problems with ProtoTour. Of the 11 participants in the experimental evaluation described in the next chapter, none of them expressed confusion over distinguishing ProtoTour from the ELDER visual prototype.

3.6 Summary

This chapter has described the design of ProtoTour using the ELDER visual prototype as an example. Functionality of ProtoTour meets the requirements for a prototype-centred explanation tool specified in the 'Final Discussion and Conclusions' of chapter 2. ProtoTour aims to exploit the potential of visual prototypes for facilitating the comprehension of HCI design intent within a software team, by addressing the deficiencies of using prototypes in this way. Primarily, ProtoTour is intended to provide team members with a better explanation of the software than a visual prototype could. There are two main ways that it aims to achieve this; firstly, by reducing the ambiguity of visual prototypes through the provision of further targeted explanation and clarification; secondly, by providing guided 'tours' of prototypes to improve their accessibility.

The latter sections of the chapter indicated that ProtoTour explanations of a visual prototype are unlikely to be as rapidly produced as the prototypes they explain. However, significantly better tools are available than were used in the production of this first prototype of ProtoTour.

Finally, a semi-formal formative evaluation of ProtoTour is described along with the important changes made as a result.

3.6.1 The Next Stage in the Research

The next stage of this research aims to test the utility of ProtoTour with commercial programmers. The ProtoTour representation of the ELDER visual prototype described in this chapter formed the basis of the experiment described in the following chapter.

Chapter 4 **Evaluation of the Utility of the ProtoTour Concept**

Chapter 4 Evaluation of the Utility of the ProtoTour Concept

4.1 Introduction

The qualitative investigation described in chapter 2 found prototypes developed by the HCI designer to be an effective medium of communication and means of facilitating comprehension (particularly of HCI design intent) within a commercial software development team. However, this use of prototypes was also found to be flawed. The main flaws discovered were ambiguity, and inaccessibility. Other problems discovered included inability to communicate rationale underlying the design, and similarly, the lack of general understanding about the project, its objectives and the application area. These problems meant that programmers were not able to effectively contribute design alternatives, or negotiate over functionality. Furthermore, a high level of communication between the HCI designer and the programmers was required in order to maintain conceptual integrity of design and to compensate for the flaws and problems inherent in using visual prototypes.

A prototype-centred explanation tool was specified in chapter 2 to address the flaws in the use of prototyping as a medium of communication and as a means of facilitating comprehension within a software team. Chapter 3 describes the design and implementation of one such tool which has been called ProtoTour.

This chapter describes an evaluation of the potential **utility** of the ProtoTour concept in a simulated commercial software development situation. The aim was to discover whether ProtoTour could help programmers better understand HCI design intent reflected in the prototype and whether this would reduce the burden of explanation on the HCI designer.

The evaluation did not seek to analyse the viability of producing a ProtoTour representation of an existing prototype.

4.2 Research Design

This section introduces the research questions. Literature related to carrying out research into HCI and aspects of software production is then revisited. Drawing on the literature, characteristics required of a research design are outlined. Finally, candidate research designs are evaluated. A quantitative experimental design is ultimately selected as the most appropriate means of answering the research questions, although even this has an exploratory element to it.

4.2.1 The Four Research Questions

The following research questions were posed to discover whether ProtoTour, as an intervention in a commercial software development situation, could help programmers better understand HCI design intent reflected in the prototype and reduce the burden of explanation on the HCI designer:

1. Does ProtoTour convey a better understanding of HCI design intent (i.e. less ambiguous) than a visual prototype?

2. Does ProtoTour improve the accessibility of the visual prototype?

3. Does the provision of extra information in ProtoTour (including design rationale) enable programmers to offer improved implementation alternatives than if they just had access to a visual prototype and an HCI designer?

4. Does ProtoTour reduce the amount of time that HCI designers need to spend explaining aspects of the visual prototype to programmers at their request?

4.2.2 Literature Relating to Conducting Research into HCI and Aspects of Software Production

An essential characteristic of this research design is the need to assess the ProtoTour intervention in a commercial software context with computer programmers. From the literature, it is apparent that conducting research into aspects of software production is fraught with difficulty. Severe individual differences between programmers in empirical studies are the norm. Rigorous assessment of the effects of a new technique on the software production process is notoriously difficult. This section briefly revisits the literature in these areas.

4.2.2.1 Individual Differences

Sheil (1987) found that the high degree of variability among programmers made simple experimental design prone to negative conclusions as slight systematic effects were “washed out” by large within groups variation. Therefore, he recommended that studies should report observed variability accounted for. In his, 'substantiating programmer variability' paper, Curtis (1981) concluded that he continues to wrestle with the problem of individual differences among programmers, which disguise systematic effects in experimental research. Brooks (1980) also cited the individual differences problem and pointed out that it is not possible to stratify participants according to ability as it is difficult to find an appropriate measure. Ultimately, he recommended the use of repeated experimental measures designs where possible.

There is no reason to think that individual differences amongst professionals in the software industry is confined to computer programmers. Although traditionally many roles in a software team stem from programming, with the advent of multi-disciplinary teams it is likely that individual differences among professionals from other fields will also become part of the research challenge.

4.2.2.2 Assessment of a New Technique or Method

Weinberg (1971) predicted that meaningful studies of programming teams would be "difficult at best". Consequently, when trying to assess the impact of a new technique or method on a software development team, this is just the starting point.

From the field of HCI, Bellotti (1990) and Carey et al. (1991) agreed that measuring how a new technique impacts the design process is notoriously difficult. Similarly, Harker (1991) found that the direct assessment of the benefits of prototyping would be hard and that assessment must be inferred. Kieras (1988) found that there was no feasible way to carry out laboratory research to test the methodology he presented, nor had there been opportunities to systematically evaluate the usefulness of any design tools recently proposed. Similar difficulties were reported by Eberts and Brock (1988), when attempting to measure the effectiveness of Computer-Aided Instruction (CAI) software. Curtis et al. (1986) also found laboratory experiments in software psychology to be impossible economically due to the nature of technology and the preclusively high cost of programmers. The root of these experimental problems is that it is often almost impossible to conceive an experiment that facilitates a rigorous comparison of a new technique with an old technique. If a software team use a development methodology on one project, it is impossible to re-create identical conditions to test an alternative methodology. One of the main reasons for this is the lack of homogeneity among professional computer programmers.

Kieras (1988) ultimately concluded that the only way to find out if his methodology worked would be to try it out on actual design problems. Clearly, this would clearly not facilitate any benchmarking of his methodology.

Sheil (1987) cautioned that the Hawthorne effect might also have an influence on studying the introduction of new technological innovation, as changes of behaviour may be caused by the study itself rather than the innovation.

One way of addressing these methodological difficulties was highly criticised by Brooks (1980). He attacked the prevalence of studies involving "beginning programmers”, the greatest source of which were students, and the tendency for studies to focus on “tiny artificial problems”. However, he did acknowledge that the preparation of stimulus material for studies was difficult. Sheil’s (1987) paper emphasises the general difficulty of designing an experiment that will generalise to a real world situation.

4.2.2.3 Summary

Individual differences between programmers has long been a difficult issue for research into aspects of software production. Early experiments involving the student programmers working on artificial problems has been much criticised, but the preclusive cost of commercial programmers remains. Furthermore, generalising from artificial experiments to practical software production situations has proved difficult. This has led some researchers to focus on studying software production in practice (e.g. through participant-observation) rather than devising experiments; whereas others call for more practitioner case studies to be published.

4.2.3 Characteristics of the Research Design

This section outlines the required characteristics that the research design must address.

The research questions aim to evaluate whether the concept of a ProtoTour style intervention would be an improvement over utilisation of typical visual prototypes (or, “chauffeured prototypes”, Preece et. al, 1994) for conveying HCI design intent within a software team. Therefore, an essential characteristic of the research design is that it must facilitate this **comparison**.

The results of this research must be applicable in the real world of software development. Much HCI research is deemed irrelevant or inappropriate to commercial software practice (see section 1.9), seriously calling into question its validity. Therefore, another key characteristic of the research design is the need to be **commercially grounded**. This meant that the evaluation itself had to be close to commercial realities so that the results would be of commercial relevance.

Criticisms of other studies of commercial software practices have centred around the use of novice programmers and students as participants, because they are unrepresentative of commercial computer programmers. Maintaining a genuine commercial grounding was of central importance to this research design. Therefore, **the sample had to come from the population of commercial computer programmers**. The cost of commercial programmers’ time was therefore a serious constraint. Consequently, **either the research design would have to utilise only a small amount of each participant’s time, or it would have to find a passive, non-disruptive means of studying them at work.**

The decision to draw the sample from the population of commercial programmers presents another difficulty. Individual differences amongst programmers is well recognised and it is difficult to design out of experiments involving computer programmers. For example, **it is not deemed feasible to allocate programmers to experimental groups on the basis of some measurement of their ability** – appropriate measures for this have not been developed. Attempting to divide programmers into experimental groups is also contrary to commercial software reality, where it is strongly believed that the prevalence of mixed ability teams is the direct consequence of individual differences findings.

Because the ProtoTour style intervention is a new concept, it would **not be possible to study existing commercial software teams already utilising similar ideas**. Similarly, it was **unlikely that a software project manager would welcome it on an experimental basis;** doing so would inevitable increase the level of risk associated with their project.

Specifics of the research questions presented further difficulties. Research question one demanded an **assessment of programmers' understanding of HCI design intent**. Question two aimed to **assess accessibility of the intervention**; question three to **assess the quality of implementation alternatives** suggested by programmers. Research question four aimed to **assess the level of time that HCI designers would need to spend explaining the intervention** to programmers.

4.2.4 Selection of an Experimental Research Design

This section describes the experimental research design selected and other candidates that were considered.

4.2.4.1 Candidate 1 – Qualitative Study of a 'live' Trial of ProtoTour

The first research design considered was a 'live' trial of the ProtoTour intervention. It was clear that a qualitative design based on a single software project would not adequately address the research questions according to the required research design characteristics. The primary shortcoming of this design would be the lack of a valid comparison between the ProtoTour intervention and the visual prototype intervention. Awaiting an appropriate software project opportunity to arise with the software team studied previously (i.e. described in chapter 2) was considered, but there were a number of problems with this approach in practice. This are listed below.

- Projects studied were long term (year-long) and new projects would be of a similar duration. Therefore, the basis for comparison would be participants' memories. Over such a duration, experiential effects were believed likely to have too severe an effect amid the complexities of a software development to afford a useful comparison.

- Studying a further project with the same software team would not have been practical due to changes in the composition of the team over time. Individual team members potentially have a large effect on the overall software project, so changes in the composition of the team would introduce a serious confound to the study.

- The ProtoTour intervention itself was only a prototype which would make a 'live' trial problematic.

- A suitable commercial project was not available for such a 'live' trail and waiting for a project would have caused an unacceptable delay to the research.

4.2.4.2 Candidate 2 – Qualitative Study of Two 'live' Trials

In an attempt to address some of the problems with the 'live' trial of ProtoTour, the concept of a comparison of the two approaches on 'live' projects was considered. On the surface, it appeared that this may produce a more valid comparison between the ProtoTour and visual prototype interventions. However, the viability of this research design was subject to further flaws:

- the likelihood of finding two projects and development teams which were sufficiently similar for a valid comparison to be made was considered almost nil;

- too many potential extraneous effects could occur within and between the two projects;

- an experiment involving two different approaches to software implementation on commercial projects, where the implication is that one approach is superior to the next, is unlikely to meet with commercial approval.

A qualitative assessment of the ProtoTour intervention on a 'live' software project would be a logical progression for the PhD research. However, this basis for a research design was deemed unsuitable, for the reasons outlined above, primarily, the lack of a valid comparison of the approaches.

4.2.4.3 Candidate 3 – Quantitative Comparison based on a Fictitious Project

Because a research design based on a 'live' project was deemed unfeasible, the benefits of a qualitative approach were reduced. A quantitative experimental design could afford valid and controllable comparisons between the ProtoTour and the visual prototype interventions.

A quantitative approach was considered particularly appropriate to the research design characteristics. Measurable assessments were required to compare: programmers' understanding of HCI design intent; the accessibility of the interventions; the quality of implementation alternatives suggested by programmers; and the amount of time that an HCI designer would spend explaining the intervention to programmers.

An experimental design would have the potential to greatly reduce effects due to individual differences amongst commercial programmers, if a large enough sample size was used.

Although this research design appears appropriate, it suffers a major flaw. The experiment would be based on a fictitious project, thus it would not be commercially realistic. Defining the project and constructing a visual prototype, and a ProtoTour intervention would also involve a large amount of development effort. However, the main drawback would be the unrealistic nature of the project, and the introduction of bias through the researcher's construction of the two interventions.

4.2.4.4 Candidate 4 (the selected design) – Quantitative Comparison Experiment Utilising an Existing Commercially Developed Visual Prototype

A compromise experimental design approach, based on the utilisation of an existing prototype, was adopted in order to provide a realistic and meaningful direct comparison between the currently used visual prototypes and the proposed ProtoTour representation. The existing visual prototype was developed by the researcher as part of the development of a software application called ELDER (completed over one year before this experiment took place). Using the ELDER prototype one experimental group of programmers was to perform a series of design tasks supported by an HCI designer, thereby closely mimicking current software development practice. The other experimental group used ProtoTour in place of the prototype to carry out the same series of design tasks, and therefore provided a simulation of potential future practice.

Thus, this quantitative experimental design addresses all of the required characteristics of the design.

- The design provides a basis for comparison of the two interventions.
- The design is commercially grounded in three ways. Firstly, through utilisation of a commercially developed visual prototype; secondly, by deriving the sample from the population of commercial computer programmers; thirdly, through devising a commercially realistic experiment design.
- The design compensates for individuals differences amongst programmers and does not require them to be divided into experimental groups based on their ability. Instead, a more commercially realistic random assignment of participants to experimental groups is intended.
- The quantitative and experimental nature of the research design facilitates the measurement required by the research questions.

There is one criticism to level at this experiment research design. Although ProtoTour and the visual prototype show exactly the same user interface of the proposed ELDER software, ProtoTour contains more information. To address this criticism, all the information available to the ProtoTour group was made available to the visual prototype group by the HCI designer (i.e. as had been the case in the projects described in chapter 2). The overriding attraction for adopting this research design is that it maps very closely to a real commercial software development scenario. It also provides a mechanism for comparison of visual prototyping and the ProtoTour interventions.

4.2.4.5 Summary

This section has described the candidate research designs and outlined the design finally selected. The required characteristics outlined in the previous section provided the reasons for the research design selection. Of the drivers, perhaps the most important was the need to compare visual prototypes and ProtoTour interventions in a commercially realistic and measurable way. Following the rejection of a 'live' trial as a viable design, a quantitative experimental approach was found to be the most appropriate means of addressing the research characteristics.

It is important to note that **there is an exploratory element to the selected experimental design**, as with qualitative studies reported in chapter 2. The ProtoTour intervention is a completely new concept as is the integration of an HCI designer role into a software team. This experiment does not add another brick to a wall of results from similar experiments, rather, this experiment could be considered the first brick in a new wall.

4.3 Method

This section describes the method adopted for the experiment based on the selected research design from the previous section. The section begins with a description of the target sample and the sampling strategy. Apparatus and material used are described, primarily focusing on the rationale for questionnaire items adopted. An overview of the procedure and the steps taken to ensure commercial realism precede a detailed description of the experimental considerations and experiment design. Finally, two pilot studies and the subsequent changes to the experiment are described.

4.3.1 Sample

The target sample was commercial programmers who had previously utilised visual prototypes as part of the software production process, particularly as a user interface design specification, and commercial programmers who were likely to be in this position in the future. It was further restricted to programmers involved in the implementation of complex graphical user interfaces. The total sample size achieved was 22 participants.

Because commercial programmers are expensive, incorporating 22 of them in an experiment lasting over one and a half hours was not straightforward. The sampling strategy was therefore opportunistic. Seven programmers came from the company that produced the software described in the qualitative investigation described in chapter 2, and nine came from the client of the project 2 software. A further six programmers came from personal contacts; two were from the same company and the remainder were from a variety of companies.

The sample could not be considered homogeneous. Some participants had worked with an HCI or UI designer before and some had not; some had worked with visual prototypes before and some had not. Individual differences in aspects of the performance of programmers is vast and well documented (see sections 1.10.2.1 and 4.2.2.1). To counter these effects, the programmers from each company were divided as equally as possible into the two experimental groups. However, individual assignments to groups were random. It was expected that programmers from the same company would have similar recent work experiences, effects of which could be removed from the experiment in this way.

Characteristics of the sample were measured through a pre-test questionnaire (reproduced in Appendix D.). These characteristics were recorded because of the partially exploratory nature of the experiment. Theory dictated that these characteristics could not be used to assign individual participants to experimental groups (Brooks, 1980, see section 4.2.2.1). However, this data was recorded in the event that experimental results indicated severe biases, which could possibly be explained by these characteristics.

Assignment of individuals to groups was essentially random, although every company was equally represented in each group.

The main demographic data describing the sample comprised the following characteristics:

- number of years of programming experience;
- number of years of experience programming user interfaces;
- prior experience of user-centred design split by employer;
- prior experience of using visual prototypes;
- familiarity with Microsoft Windows Help[1].

These characteristics of the sample are presented in results section 4.4.1.

4.3.2 Apparatus and Materials

This section describes experimental measurement instruments adopted. It then describes the rationale for questionnaire and other assessment items. Finally, a list of the apparatus and materials is included.

4.3.2.1 Experimental Measurement Instruments

The partially exploratory nature of the experiment and the diverse nature of the research questions led to the selection of four experimental measurement instruments. These instruments and their rationale are described below.

1. Participant opinion rating scales and open-ended questions

To acknowledge the exploratory nature of the experiment some pre-test opinion measures were introduced. Items included were intended for use in post hoc analysis to help explain any strong biases which may have arisen during the experiment.

Research question 2, 'Does ProtoTour improve the accessibility of the visual prototype?', was primarily assessed using rating scales. In particular, a pre-test/post-test design was utilised.

Research questions 3 and 4 were also supported by participant opinion ratings.

The exploratory nature of the experiment design introduced the concern that statistically significant results between experimental groups may not be found. Individual differences and a moderate sample size (n=22) further reduced the likelihood of significant results by quantitative methods alone. Therefore, qualitative data in the form of comments were also collected to supplement quantitative results.

[1] This additional characteristic was added following the pilot studies and was utilised in the assignment of individuals to experimental groups (see section 4.3.7.2)

2. Expert assessment of participant task performance

Expert assessment was primarily used to answer the first and third research questions: 'Does ProtoTour convey a better understanding of HCI design intent (i.e. less ambiguous) than a visual prototype?'; and 'Does the provision of extra information in ProtoTour (including design rationale) enable programmers to offer improved implementation alternatives than if they just had access to a visual prototype and an HCI designer?'

Assessing participants' solutions to the implementation tasks was thought to be an appropriate means of ascertaining the extent to which they had understood HCI design intent. In commercial practice, this would be the ultimate test of whether they had understood design intent.

Because of the diversity of possible approaches and solutions to the implementation tasks[2], it was necessary for the solutions to be assessed by a commercial software engineering expert.

3. Objective measures of task performance

A number of objective experimental measures were devised to supplement other measurement instruments. In particular, these measures removed any potential bias associated with expert assessment, whilst still providing data on the participants' performance and understanding of HCI design intent. Therefore, objective measures were utilised for research questions one and three.

Objective measures devised needed no form of subjective interpretation as part of the data collection. For example, one measure recorded the length of time in days that the participant estimated would be needed to implement a user interface design. Another, took the form of a comprehension 'test' which asked participants to annotate a diagram – the objective measure consisted of a count of the number of correct annotations based on a marking scheme.

4. Observation during the experiment

Observation was employed to address research question four: 'Does ProtoTour reduce the amount of time that HCI designers need to spend explaining aspects of the visual prototype to programmers at their request?'

Participants were encouraged to ask the experimenter questions about the implementation tasks. Observed measures recorded and categorised these questions.

[2] Curtis et al. (1987) noted the "tremendous variability" in how programmers go about tasks – section 1.10.2.1

4.3.2.2 Rationale for Questionnaire, Expert and Other Assessment Items

This section outlines the rationale for the questionnaire, expert and other assessment items. The questions/assessments are categorised by research question and experimental instrument.

4.3.2.2.1 Pre-test Items & Demographics

(Pre-test questionnaire – see Appendix D1.3)

This data was primarily recorded in the event that experimental results indicated severe biases, which could possibly be explained by these characteristics. Questions 1.1 and 1.6 were used in the assignment of participants to experimental groups. Note that Q1.7 was also used as part of a pre-test/post-test measure reported in section 4.3.2.2.3.

The following questions were utilised in the assignment of participants to experimental groups.

Q1.1 Organisation
Q1.6 Participants' Familiarity with Microsoft Windows Help

Rationale: Recent shared work experiences were thought to be a factor in participants' opinions and performance in many aspects of this experiment.

Familiarity with Windows Help was used as a secondary criteria, because participants unfamiliar with it would be at a disadvantage if assigned to Group 2 (ProtoTour).

The following questions share the rationale described below.

Q1.2 Participants' Programming Experience
Q1.3 Participants' User Interface Programming Experience
Q1.4 Participants' Prior Experience of User-Centred Design
Q1.5 Participants' Prior Experience of Using Visual Prototypes
Q1.7 Participants' Rating of the Use of Visual Prototypes
Q1.8 Participants' Rating of the User-Centred Design (or user interface issues) Compared with Functionality
Q1.9 Participants' Rating of the Importance of the Role of HCI Designer

Rationale: These measures were introduced, because they were thought to have the potential to explain any strong biases occurring in the results.

4.3.2.2.2 Research Question 1: Does ProtoTour convey a better understanding of HCI design intent (i.e. less ambiguous) than a visual prototype?

Experimental Instrument: Expert assessment of participant task performance

Q3.1b How comprehensive is the participant's preliminary design? (Participant task Q3.1 – see Appendix D1.5: Assessor Instructions – Appendix D1.10)

Rationale: The comprehensiveness of the preliminary designs would show how well the participants had understood HCI design intent.

Q3.2 Expert assessment of appropriateness of assumptions made (Participant task Q3.2 – see Appendix D1.5: Assessor Instructions – Appendix D1.11)

Rationale: This was introduced because an understanding of HCI design intent would contribute to informing the assumptions made by participants.

Q3.3 Expert assessment of appropriateness of implementation difficulties identified (Participant task Q3.3 – see Appendix D1.5: Assessor Instructions – Appendix D1.12)

Rationale: This provided a measure of degree to which participants understood the interaction complexity of the HCI design of the Checklist Window.

Q3.6a Expert assessment of how well overall, the participant appears to have understood HCI design intent (Participant task Q3.1-Q3.5 – see Appendix D1.5: Assessor Instructions – Appendix D1.15 Q3.6a)

Rationale: This was designed to provide the expert assessor with a means to express an overall impression ('gut feel') of the participant's understanding of HCI design intent.

Q3.6b Expert assessment of consistency of participants' preliminary design with HCI design intent (Participant task Q3.1-Q3.5 – see Appendix D1.5: Assessor Instructions – Appendix D1.15 Q3.6b)

Rationale: The expert's 'gut feel' for the participants' understanding may be different from their 'objective' assessment of the preliminary design produced.

Experimental Instrument: Objective measures of task performance

Q3.5 Participants estimate (in days) of how long the checklist would take to implement (Participant task Q3.5 – see Appendix D1.5)

Rationale: It was thought that participants' time estimates to implement the functionality (i.e. how long) would provide an insight into their understanding of the complexity of the HCI design presented.

Questions Q4.1-Q4.4 share rationale described below.

Q4.1 Measurement of correctness and completeness of participants' annotation of the functionality of the Checklist Window (Post-test questionnaire – see Appendix D1.7: Also, Framework for quantitative measurement of Q4.1 – see Appendix D1.16)

Q4.2 Measurement of correctness of participants' description of the relationship between the checklist and the log file (Post-test questionnaire – see Appendix D1.7)

Q4.3 Measurement of correctness of participants' description of the relationship between the checklist and the browser (Post-test questionnaire – see Appendix D1.7)

Q4.4 Measurement of correctness of participants' description of what should happen if advice from a chapter in ELDER is accessed which is different from the current equipment type (Post-test questionnaire – see Appendix D1.7)

Rationale: Participants' understanding of the Checklist Window could be measured objectively with these comprehension-style questions, thus providing an insight into their understanding of the HCI design intent.

4.3.2.2.3 Research Question 2: Does ProtoTour improve the accessibility of the visual prototype?

Experimental instrument: Participant opinion rating scales and open-ended questions

Q1.7 Pre-test / Q4.10 Post-test rating of how useful participants' perceived prototypes to be (Pre-test questionnaire – see Appendix D1.3: Post-test questionnaire – see Appendix D1.7)

Rationale: This pre/post-test measure was designed to look for an improvement in participants' opinions towards the usefulness of prototypes in software design. Although tangential to the research question, participants who value the use of prototypes would be more open to their use. This 'acceptance' measure was regarded as providing an indication of the early stages of accessibility – do the programmers think it is worth making prototypes accessible to them?

Q4.5 Participants' rating of how easy it was to visualise the software from the prototype approach adopted (Post-test questionnaire – see Appendix D1.7)

Rationale: Visualisation is a fundamental aspect of comprehension. The ease with which participants were able to visualise the software would provide an indication of how accessible they had found the prototype.

Q4.6(i) Participants' rating of the utility of supplemental information provided (Post-test questionnaire – see Appendix D1.7)

Rationale: Because supplemental information was included in ProtoTour but had to be requested in the visual prototype, this measure was introduced to assess which approach the participants found most appropriate. It was thought that participants might prefer to receive extra information from ProtoTour, rather than risking appearing foolish by verbalising an 'obvious' question.

Q4.7 Participants' rating of whether they were able to access information about WHY the software was designed as it was (Post-test questionnaire – see Appendix D1.7)

Rationale: This was necessary to ascertain whether users had successfully obtained design rationale from the HCI designer or from ProtoTour. This experimental measure therefore aimed to compare the accessibility of design rationale information between the experimental groups.

Q.12 Participants' rating of whether they would like to see prototypes used in a similar way again (Post-test questionnaire – see Appendix D1.7)

Rationale: This experimental measure was **not designed to make comparisons** between experimental groups, because their participants used different interventions. However, both interventions were based on the use of a visual prototype within software design and implementation. It was therefore appropriate to assess whether participants favoured these interventions (compared with an implied alternative of not using them). This would indicate whether such approaches would be favourable and would consequently whether they would become accessible, through demand.

4.3.2.2.4 *Research Question 3: Does the provision of extra information in ProtoTour (including design rationale) enable programmers to offer improved implementation alternatives than if they just had access to a visual prototype and an HCI designer?*

Experimental instrument: Expert assessment of participant task performance

Q3.4a Expert assessment of implementation alternatives identified (Participant task Q3.4 – see Appendix D1.5: Assessor Instructions – Appendix D1.13)

Rationale: It was believed that the most appropriate implementation alternatives would come from participants with the best understanding of HCI design intent.

Q3.4b Expert assessment of participants' assessments of the likely impact on usability that the implementation alternatives will have (Participant task Q3.4 – see Appendix D1.5: Assessor Instructions – Appendix D1.14)

Rationale: The two interventions were believed to have differing potential for informing participants about usability. This would be revealed through participants' assessments of the usability impact of the alternatives they suggested.

Q3.4c Expert assessment of the participants' assessment of the implementation alternatives likely impact on implementation (Participant task Q3.4 – see Appendix D1.5: Assessor Instructions – Appendix D1.14)

Rationale: This was necessary to ascertain whether the participants' suggested alternatives would genuinely improve the implementation.

Experimental Instrument: Objective measures of task performance

Objective measures 3.4a(i)-(iii) have similar rationale, described below.

Q3.4a(i) Number of implementation alternatives suggested (Participant task Q3.4 – see Appendix D1.5)

Q3.4a(ii) Number of alternatives suggested primarily relating to usability (Participant task Q3.4 – see Appendix D1.5)

Q3.4a(iii) Number of alternatives suggested primarily relating to implementation (Participant task Q3.4 – see Appendix D1.5: Assessor Instructions – Appendix D1.14)

Rationale: The interventions were thought to have differing potential for prompting programmers to consider implementation alternatives, thus the number from each group was measured. Similarly, it was thought likely that differences may also be apparent in the number of alternatives suggested relating to usability and the number relating to implementation.

For example, ProtoTour proactively provided programmers with the 'big picture' of the UI design, explaining details of users and tasks. This was considered a potential means of elevating the importance of usability considerations within the minds of programmers. Thus, programmers made more aware of the 'big picture' may suggest more alternatives relating to usability.

Experimental instrument: Participant opinion rating scales and open-ended questions

Objective measures Q4.8-Q4.9 have similar rationale, described below.

Q4.8 Participants' self rating of how useful design rationale information was in the identification of implementation alternatives (Post-test questionnaire – see Appendix D1.7)

Q4.9 Participants' self rating how useful design rationale was in assessing the impact of alternatives on usability (Post-test questionnaire – see Appendix D1.7)

Rationale: These measures assessed whether participants considered design rationale to have had a bearing on the alternatives their suggested alternatives. This was necessary, because the usefulness of design rationale is questionable.

4.3.2.2.5 Research Question 4: Does ProtoTour reduce the amount of time that HCI Designers need to spend explaining aspects of the visual prototype to programmers at their request?

Experimental instrument: Observation during the experiment

Obs 3(i) Number of questions participants' asked concerning the usability of prototype (Group 1) or ProtoTour (Group 2) (Observation record – see Appendix D1.5)

Rationale: This was necessary in order to the hypothesis that ProtoTour is easier to use than the traditional visual prototype.

Obs 3(ii) Number of questions participants' asked about the intended functionality of the software portrayed in the prototype (Observation record – see Appendix D1.5)

Rationale: This measure assessed whether the intervention had an effect on the number of questions that participants asked regarding intended functionality (HCI design intent).

Obs 3(iii) Number of questions participants' asked about the rationale underlying the user interface design presented in the prototype (Observation record – see Appendix D1.5)

Rationale: This assessed whether the proactive provision of design rationale within ProtoTour, resulted in participants asking less questions about this, than those using the prototype intervention where such information could only be obtained by request.

Obs 3(iv) Number of questions participants' asked about the intended users or their tasks (Observation record – see Appendix D1.5)

Rationale: This measure assessed whether the proactive provision of information about users and tasks within ProtoTour, resulted in participants asking less questions about this, than those using the prototype intervention where such information could only be obtained by request.

Obs 3(v) Number of questions participants' asked about the scope of the design task they had been set (Observation record – see Appendix D1.5)

Rationale: This observation was devised as an experimental check, making use of the fact that participants of both experimental groups received almost identical instructions. If there were significant differences between experimental groups on this measure, it would provide an indication that participants of one group had a greater propensity to ask questions than those of the other group.

Experimental instrument: Participant opinion rating scales and open-ended questions

Q4.6(iii) Participants' rating of how useful the HCI designer was during the exercise
(Post-test questionnaire – see Appendix D1.7)

Rationale: It was anticipated that participants using ProtoTour would have less need to ask questions of the HCI designer and would therefore indicate that they found him redundant during the experiment.

4.3.2.2.6 Other Results - Summary of Participants' Comments and Criticisms

A single questionnaire item asked participants for comments and criticisms about the intervention they experienced.

Experimental instrument: Participant opinion rating scales and open-ended questions

Q4.11 Participants comments and criticisms of the use of the intervention they experienced during the experiment, as a vehicle for 'hand-over' of the outward design of the software (Post-test questionnaire – see Appendix D1.7)

Rationale: Because the experiment was partially exploratory, a qualitative item was used to collect participants' views about the intervention that they had experienced during the experiment.

4.3.2.3 List of Apparatus and Materials used in the experiment

The following apparatus and materials were used in the experiment:

Apparatus
486 PC running Microsoft Windows 3.11
ELDER prototype (written in Visual Basic 3.0)
ProtoTour representation of the ELDER prototype (described in chapter 3)
Various writing implements (pens, pencils and rulers) and a flowchart template

Materials
Instructions for participants of both experimental groups (Appendix D1.2)
Pre-test questionnaire (Appendix D1.3)
Script for presentation of the ELDER prototype (for the use of the experimenter - Appendix D1.4)
Description of preliminary design tasks for both experimental groups (Appendix D1.5)
Observation table (for experimenter to record questions asked - Appendix D1.6)
Post-test questionnaire for both experimental groups (Appendix D1.7)
Instructions for the assessor (Appendix D1.9)
Frameworks for assessing participant responses to each design task (Appendix D1.10-1.15)
'Model Answers' to the preliminary design question (Appendix D1.10.1)
Framework for measuring the of correctness of the annotation of the Checklist Window (Appendix D1.16)

4.3.3 Overview of Procedure

A sample comprising commercial software developers was divided into two experimental groups. Participants from the same company were split evenly between the groups; but were otherwise randomly allocated. After completing a pre-test questionnaire, all participants received a five minute presentation of the ELDER prototype. Group 1 participants were then given the ELDER prototype, whilst Group 2 participants were given a ProtoTour representation of the ELDER prototype. Each group spent the first 15 minutes investigating the intended functionality of the ultimate ELDER end product (HCI design intent) illustrated by the given representation. The participants then spent a further 45 minutes completing the set design tasks, referring to the representation or the HCI designer as necessary. A post-test questionnaire was then completed to assess the participants' understanding of the intended functionality of the ELDER end product (without further reference to any previous material) and their opinion of the representation they had used.

4.3.4 Overview of Steps Taken to Ensure Commercial Realism of Experiment

To test the utility of the ProtoTour concept, the experiment had to realistically reproduce a commercial software development situation. This realism was achieved in several ways. Firstly, the prototype used as the subject of the experiment (i.e. the prototype which was the subject of the ProtoTour representation) was produced by the experimenter as part of a separate commercial software development. The decision to use this prototype (ELDER) was made after its completion and no part of the prototype was adapted for the experiment. Secondly, the programmers used in the experiment were all currently employed commercial software developers. Thirdly, the design tasks were carefully conceived in consultation with senior software engineers to ensure that they were typical of the tasks that commercial software developers are confronted with. Fourthly, the experiment was carried out under a strict time constraint (commercial software developers' time is expensive and their work is often influenced by this). Fifthly, the experiment was carried out at the participants' place of work. Finally, the expert recruited to assess the participants' design tasks had been involved with the ELDER project and shared the conceptualisation of the ELDER product.

4.3.5 Experimental Design Considerations

This section outlines experimental constraints, methodological considerations and data analysis considerations.

4.3.5.1 Experimental Constraints

In addition to the experimental constraints outlined in section 4.3.4 (i.e. that experiments would be carried out in a commercial environment, and that the duration of each experiment would be strictly limited due to time and the cost of participants), a further constraint was experienced. Experiments had to be carried out in a large number of locations with very little time to set up equipment.

4.3.5.2 Methodological Considerations

The main methodological considerations made during the design of the experiment are outlined in this section.

Implications of Experimental Constraints

Participants' availability restricted the duration of the experiment to 1½ hours. Such time constraints were preclusive to carrying out analysis of the programmers' personal characteristics, thus leaving individual differences between programmers as a potential experimental confound (addressed in part by random assignment of participants to experimental groups).

Individual Differences between Programmers

Individual differences in the performance of computer programmers is a well documented phenomena, which was addressed by the strategy used to assign participants to experimental groups (see sections 1.10.2.1 and 4.2.2.1). Two strategies were considered to address this issue, either the programmers could be assigned according to various personal characteristics or simple random assignment could be used. The complexity and diversity of the variables leading to individual differences between programmers pointed to random assignment of participants throughout the study. Because the results of the evaluation needed to be generalised to commercial software development teams where individual differences exist and teams are often mixed ability[3], it would not have been useful for the results of the evaluation to show that ProtoTour was better for programmers of a certain intelligence with a certain level of conceptualisation ability.

One improvement to the basic random assignment of participants to experimental groups involved taking account of their current work experiences. Participants in the sample were not all independent individuals drawn from different companies and contexts. A large proportion of the sample consisted of groups of programmers taken from various companies. These programmers were likely to share experiences and work contexts which may have caused them to formulate similar attitudes or gain similar skills. Thus, within the context of random assignment, participants sharing the same work contexts were split evenly between the groups.

In order for the random assignment of participants to experimental groups to eliminate effects due to individual differences a large sample size was required. The intended sample size, although reasonably large (in excess of 20 participants), was considered unlikely to be completely effective at cancelling out individual difference confounds. To

[3] Results from the qualitative investigation described in chapter 2 suggested that in commercial software practice, the team is selected primarily due to resource constraints. Whilst it may be possible to select a team of people who would not have any need for ProtoTour (perhaps super-conceptualisers, excellent communicators, high intelligence), or a team of people all of which would be likely to benefit from the use of ProtoTour, the likelihood is that the team will consist of a mixture of such people.

support quantitative experimental measures, a ‘comments and criticisms’ section was added to the post-test questionnaire to collect some qualitative data on participants’ thoughts about the use of the visual prototype and ProtoTour.

Serious consideration was given to devising a repeated-measures experiment in which the effects of individual differences could be eliminated by using each participant as their own control. However, this experimental design was ultimately considered too complex, lengthy and unrealistic.

Potential Bias from the Selection of the Visual Prototype Example Selected

The utilisation of an existing prototype (without modification), eliminated any potential bias which could have been introduced by using a hypothetical visual prototype solely for the purpose of this experiment. A hypothetical visual prototype could have been constructed to deliberately frame ProtoTour in a positive light. The use of the existing ELDER visual prototype also tested whether the construction of a ProtoTour explanation could be achieved on a real project. The completion of a ProtoTour representation of the ELDER prototype clearly proved that this was the case (see the design and implementation of ProtoTour in chapter 3). It should also be noted that no members of the sample had prior exposure to the ELDER project or the ELDER visual prototype, which was used for the experiment.

4.3.5.3 Data Analysis Considerations

The experiment described in this chapter has a broad scope and could be considered partly exploratory. At the outset, it was not known which aspects of ProtoTour would be likely to show experimental effects on which experimental measures. Consequently, a large number of research measures were employed, although each was directly associated with one of the four research questions. Analysing such results required the use of a large number of *t*-tests; the cumulative effects of which increased the chance of making type 1 errors. Multivariate statistics were not a viable option as the participants to variables ratio was a long way short of the 20:1 suggested by Tabachnick and Fidell (1983). Because of the partially exploratory nature of the experiment **all *t*-tests reported are two-tailed significance tests.** To account for the large number of *t*-tests, results significant at the $\alpha < 0.05$ level are considered marginal. Truly significant findings in this experiment are considered to be at α levels of <0.01.

For each *t*-test, normality and homogeneity of variance assumptions were considered. However, according to Hays (1988), even quite severe departures from normality make little practical difference to the conclusions of a *t*-test. Hays found the homogeneity of variance assumption to be more important than the normality assumption. Although he concluded that “for samples of equal size, relatively big differences in the population variances seem to have relatively small consequences for the conclusions derived from a *t* test... ...when in doubt use samples of the same size, or very nearly so.”(p. 303).

Therefore, in this experiment only Levene’s test of homogeneity of variances is reported. Where significant differences in variances are found, some Box-plots have been used to illustrate them. If the homogeneity of variance assumption has been violated, the *t*-test was considered to be largely valid as long as n1=n2. Tests of normality (e.g. Kurtosis and Skew) are not reported.

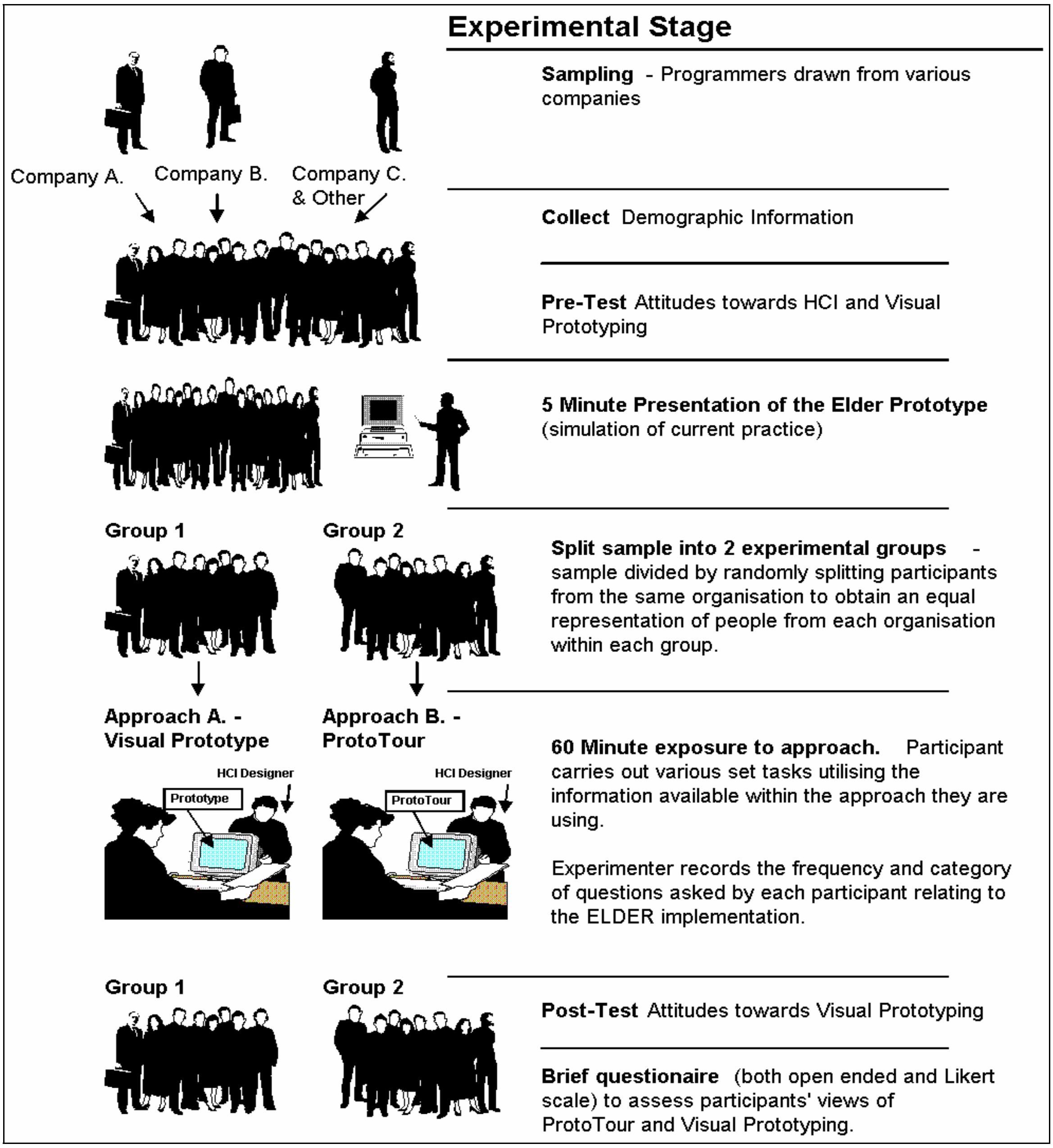
For each significant *t*-test result, the ω^2 estimate of the variance in the experimental measure accounted for by the participants' utilisation of ProtoTour or the prototype was calculated using the following equation:

$$\omega^2 = \frac{t^2 - 1}{t^2 + N_1 + N_2 - 1}$$

Using the ω^2 estimate, the power of the *t*-tests was also derived using Hays' (1988) statistical table XII, "Approximate power of the ANOVA F test, for different true values of omega-square, for different numbers of groups..." (p. 951).

4.3.6 Detailed Experimental Design

An independent groups experiment design illustrated in figure 4.1 is considered most appropriate to evaluate the potential utility of the ProtoTour concept.



Experimental Stage

Sampling - Programmers drawn from various companies

Collect Demographic Information

Pre-Test Attitudes towards HCI and Visual Prototyping

5 Minute Presentation of the Elder Prototype (simulation of current practice)

Split sample into 2 experimental groups - sample divided by randomly splitting participants from the same organisation to obtain an equal representation of people from each organisation within each group.

60 Minute exposure to approach. Participant carries out various set tasks utilising the information available within the approach they are using.

Experimenter records the frequency and category of questions asked by each participant relating to the ELDER implementation.

Post-Test Attitudes towards Visual Prototyping

Brief questionaire (both open ended and Likert scale) to assess participants' views of ProtoTour and Visual Prototyping.

Figure 4.1 The independent groups experiment design

The first experimental group performed the design tasks using a simulation of current practice involving the use of a visual prototype, a written specification and access to an HCI designer. The second experimental group carried out a simulation of potential future practices using a ProtoTour representation of the visual prototype, a written specification and access to an HCI designer.

Experimental steps are listed below:

1. Participants completed a simple questionnaire (ratings scales) prior to the study to assess their opinions of user-centred design, the use of visual prototypes in software development and their views on the presence of an HCI designer in a software team. Some simple demographic information was also collected at this stage including number of years programming experience and prior experience of user-centred design, prototyping, user interface coding experience, and familiarity with Windows help.

   Each participant was randomly assigned to an experimental group, but each group had an approximately equal representation of participants from each company.

2. Regardless of experimental group, each participant received a five minute presentation of the ELDER software through demonstration of the existing ELDER visual prototype (strictly scripted to eliminate bias).

3. The participant was then asked to complete certain tasks relating to the implementation of the ELDER software (60 minutes).

   Participants from experimental Group 1 were given the prototype, a specification document and access to the HCI designer to assist with completing their tasks. Participants from experimental Group 2 were provided with the ProtoTour representation, a specification document and access to the HCI designer to assist with completing their tasks.

   The frequency and category of questions asked by the participants during the completion of the design tasks was recorded by the experimenter.

4. After completion of tasks, participants completed a questionnaire. The questionnaire was designed to test their understanding of the HCI design intent (including proposed functionality) of the Checklist Window, which they had been working on. It also sought their opinions on various aspects of the approach they had used (i.e. visual prototype/ProtoTour). Finally, participants were asked for comments and criticisms about the approach they had used.

5. The preliminary design task in stage 3 of the experiment was primarily expert assessed. To facilitate this assessment the assessor was provided with instructions and guidelines. Because many aspects of the participants' design work were rated, a Cooper-Harper[4] style of scale was developed to assure the reliability of the expert's assessment. Model answers were also produced as a benchmark for the expert to use

[4] Cooper and Harper (1969) developed a framework for guiding assessors completion of rating scale measurements in order to improve the reliability of the assessor.

in assessing the participants' designs. This assessment material is reproduced in Appendix D.

The expert selected to assess the participants' design tasks had been involved in the ELDER project and shared the conceptualisation of the ELDER product for which the prototype is merely a concretisation. This gave the assessor a sufficiently broad understanding of the software being portrayed by the visual prototype and ProtoTour to be able to assess the correctness of participants' interpretations. The assessor was a senior software engineer.

4.3.7 Pilot Studies

Two pilot studies were necessary due to the considerable amount of changes made following the first pilot study.

4.3.7.1 Pilot Study One

An initial pilot of the experiment was carried out with three employees from the company was involved in the qualitative investigation described in chapter 2. They were not considered suitable to form part of the sample for the actual experiment as they were not full-time computer programmers. However, they carried out a degree of commercial computer programming as part of their roles and therefore knew how to program. For example, one pilot study participant was a mathematical modeller whose primary role was the development of mathematical algorithms and whose secondary role involved implementing (computer programming) these algorithms as part of commercial software projects. The skills of the pilot study participants were therefore considered sufficient for the purposes of the pilot study, but inappropriate for the experiment as programming was not their main role. The premium placed on real commercial programmers meant that they could not be 'wasted', participating in a pilot study.

The pilot uncovered some problems with the wording of certain test questions all of which were changed for the second pilot. One example of this was the estimation of how long it would take to implement the aspect of ELDER participants had been working on (estimating formed part of the role of all pilot study participants). None of the pilot study participants were happy about answering this question and all had to be persuaded to do so. This was an unsurprising finding as programmers (and others) are often reluctant to make such estimates, because they do not like to commit themselves. In the pilots there was certainly not time to produce accurate estimates so participants preferred to abstain rather than make a wild guess. Therefore the estimation task was re-worded. Furthermore, the simplification of the overall design task (although not purely for this reason) was hoped to encourage participants to make an estimate.

After running the pilot, the major concern was that participants made only scant use of the visual prototype or ProtoTour in carrying out their design tasks. Short interviews of the pilot participants established several reasons why this occurred.

1. The HCI Designer's presentation of the ELDER visual prototype was too good

The presentation of the ELDER visual prototype was apparently too comprehensive as it allowed participants to attempt the implementation tasks from what they remembered of the presentation. Their only apparent use of the visual prototype or ProtoTour was to display a single fixed screen image of the Window to be implemented to act as a reminder. Thus, the experiment was not working. Furthermore, the experiment format also presented a poor simulation of current practice. In current practice, the HCI designer is likely to present (demonstrate/walkthrough) the visual prototype to the programming team some time (say from one to six weeks) before they begin to implement the software. In the experiment, the visual prototype was presented immediately participants were asked to complete implementation tasks. This is very rarely the case in practice.

2. The aspect of Elder which participants were asked to design was too complex

The main Hypertext Information Area Window was initially selected as the aspect of ELDER which should be designed. This window was central to the ELDER application and proved too complex for the pilot study participants to design within the constraints of the experiment; the resulting designs were superficial. The Low Level Design of the Hypertext Information Area was not obvious and High Level Design was fairly trite (i.e. most people just sketched what was on screen).

3. There were too many aspects to the design tasks

There were too many design tasks set for the participants to complete within the constraints of the experiment. For participants to have completed these tasks well, they would have needed far more time and a greater amount of experience with Windows implementation, including experience of a variety of software tools and programming languages. Invariably participants selected a question they felt they could answer well, rather than completing all of the questions. This was an understandable response.

4. The participants were overloaded with information

Group 1 participants had a visual prototype and a specification document to work with, as well as availability of an HCI designer for questioning. The HCI designer had a comprehensive printout of ALL information concerning ELDER (available within the ProtoTour representation), which could be provided to the participants if they requested more information. Group 2 participants had a ProtoTour representation of the visual prototype as well as a specification document and the availability of an HCI designer for questioning. This appeared to be too much information for participants to deal with.

5. Unfamiliarity with ProtoTour

It was observed that participant unfamiliarity with ProtoTour was a potential problem. Learning how to use it seemed to add a further burden to Group 2 participants.

4.3.7.2 Changes to the Experiment Following the First Pilot Study

Aside from the changes to the wording of certain questions and instructions in the experimental material, the following changes were made to the experiment:

1. The HCI Designer's presentation of the ELDER visual prototype simplified

The presentation of the ELDER visual prototype was shortened and simplified to better mimic current practice. This also forced participants to make use of the visual prototype/ProtoTour for more detailed information.

2. The aspect of ELDER which participants were asked to design was changed

On the advice of senior software engineers, the aspect of ELDER which participants were asked to design was changed to the Checklist Window. This was considerably simpler than the Hypertext Information Area Window.

Senior software engineers suggested that participants should be asked to actually program part of ELDER as at this detailed level all things become concrete (including assumptions and mistakes). This would enable the experiment to capture the participants' suggested designs and implementations more clearly, for a better evaluation. Unfortunately, the strict time constraint on the experiment did not allow actual programming (and the logistics of programming languages and tools would be extremely troublesome). However, on the suggestion of these software engineers, the aspect of functionality which participants were asked to work with was simplified. The major benefit of this was that participants could now concentrate on Low Level Design in a format which is very close to actual programming code. Whilst this is a good compromise, it is extremely difficult to judge exactly how hard the implementation task should be to discriminate between the experimental groups. Too simple an aspect of functionality for example would enable participants in each group to perform equally well. Conversely, too difficult an aspect of functionality with such a tight time limit may see participants in each group performing equally badly. Ultimately, it was decided that the Checklist Window within ELDER would be the aspect of functionality that each group would work with. However, the uncertainty about the effectiveness of this demanded a further pilot study be carried out.

3. The number of aspects to the design tasks was reduced

The number of aspects to the design tasks was reduced by about a third. This made it feasible for the tasks to be completed in around 45 minutes instead of the 60 minutes originally allocated. Participants were instead asked to spend the first 15 minutes of the experiment (following the presentation of the visual prototype) using either the visual prototype or ProtoTour to gain a better understanding of the HCI design intent and proposed functionality of the Checklist Window BEFORE attempting the design tasks.

4. The amount of information given to participants was reduced

The printout of information from ProtoTour which the HCI designer was to make available to Group 1 participants on request was removed from the experiment. The experiment was further simplified by removing the specification document from both experimental groups.

5. Test participants familiarity with ProtoTour-like Windows' Help

It was felt that participants who were unfamiliar with Microsoft Windows Help style hypertext applications would be at a disadvantage using the ProtoTour tool. Therefore a question regarding the participants' familiarity with Windows Help was added to the pre-test questionnaire. Where possible, participants scoring low on this question were assigned to experimental Group 1. This was considered valid in the experimental context, because it was the potential UTILITY and not the USABILITY of ProtoTour which was being evaluated.

4.3.7.3 Pilot Study Two

There were two participants in the second pilot study, one of whom had participated in the initial pilot. Experimentally this was not regarded as a problem, because the focus of the experiment was sufficiently different for transfer effects to be a minor problem).

As in the first pilot, several minor changes to the wording of several questions were necessary.

During the pilot, participants seemed willing to spend 15 minutes utilising the visual prototype or ProtoTour to find out about the HCI design intent and functionality of the Checklist Window. Following this initial exposure, participants also made some use of the visual prototype or ProtoTour during the completion of the design tasks.

The scope of the design task now appeared to be more reasonable. However, participants still appeared to experience difficulties in deciding what level of detail they should show in their design. This is unsurprising, and probably unavoidable, as many programmers would not bother to represent the design at this level, preferring to begin writing the code. Both participants were able to attempt all the tasks in reasonable detail. The person repeating the pilot commented that the design task was now “much better” than the one in the initial pilot.

No major changes were necessary to the experimental design following this second pilot study.

4.4 Results

Characteristics of the sample of commercial computer programmers are first described. Results relating to each research question are then presented. Qualitative results, additional to the research questions, follow.

4.4.1 Characteristics of the Sample

Characteristics of the sample of commercial computer programmers are described in this section as they were assigned to experimental groups.

Q1.1 Organisation

Table 4.1 Cross-tabulation of participants' organisation by experimental group

COUNT	Company A	Company B	Company C	Other	Row Total
Group 1	3	4	1	3	11 50.0
Group 2	4	5	1	1	11 50.0
Column Total	7 31.8	9 40.9	2 9.1	4 18.2	22 100.0

Q1.2 Participants' Programming Experience

Table 4.2 Cross-tabulation of participants' years of programming experience by experimental group

COUNT	<1	1-2	2-5	5-9	>9	Row Total
Group 1	1	3	1	3	3	11 50.0
Group 2	1	1	3	3	3	11 50.0
Column Total	2 9.1	4 18.2	4 18.2	6 27.3	6 27.3	22 100.0

It can be seen from table 4.2 that over 55% of participants had more than five years commercial programming experience and only around 27% had less than two years experience.

Q1.3 Participants' User Interface Programming Experience

Table 4.3 Cross-tabulation of participant's years of experience programming user interfaces by experimental group

COUNT	<1	1-2	2-5	5-9	>9	Row Total
Group 1	2	2	3	3	1	11 50.0
Group 2	3	3	2	1	2	11 50.0
Column Total	5 22.7	5 22.7	5 22.7	4 18.2	3 13.6	22 100.0

From table 4.3, it can be seen that over 32% of participants had more than five years experience programming aspects of user interfaces and around 45% had less than two years experience.

Q1.4 Participants' Prior Experience of User-Centred Design

Table 4.4 Cross-tabulation of participants' prior experience of user-centred design by experimental group

COUNT	No Prior Experience of UCD	Prior Experience of UCD	Row Total
Group 1	5	6	11 50.0
Group 2	2	9	11 50.0
Column Total	7 31.8	15 68.2	22 100.0

From table 4.4, it can be seen that 68% of participants had some prior experience of user-centred design.

Q1.5 Participants' Prior Experience of Using Visual Prototypes

Table 4.5 Cross-tabulation of participants' prior experience of using visual prototypes by experimental group

COUNT	No Prior Experience of Visual Prototypes	Prior Experience of Visual Prototypes	Row Total
Group 1	2	9	11 50.0
Group 2	2	9	11 50.0
Column Total	4 18.2	18 81.8	22 100.0

From table 4.5, it is evident that almost 82% of participants had prior experience of using visual prototypes.

Q1.6 Participants' Familiarity with Microsoft Windows Help

Participants' rated their familiarity with Microsoft Windows Help on a scale of '1 - Completely Unfamiliar (never used it)' to '7 - Completely Familiar (use it regularly)'.

Levene's test led to acceptance of the hypothesis that variances are homogenous (F=.278, p=.604).

The following table summarises the results of a *t*-test.

Table 4.6 *t*-test of participants' rating of their familiarity with Microsoft Windows Help

	N	**Mean**	**SD**	**SE**	***t***	**df**	**p (1 tailed)**
Group 1	11	5.5455	1.508	.455	-1.22	20	.118
Group 2	11	6.2727	1.272	.384			

There was no significant difference between groups.

Q1.7 Participants' Rating of the Use of Visual Prototypes

Participants' rated the use of visual prototypes within software development to aid design and implementation on a scale of '1- Not Useful at All' to '7 - Extremely Useful'.

Levene's test led to the rejection of the hypothesis that variances are homogenous (F=8.431, p=.009). The following Box-Plot further illustrates this result, showing significantly greater variance in Group 2.

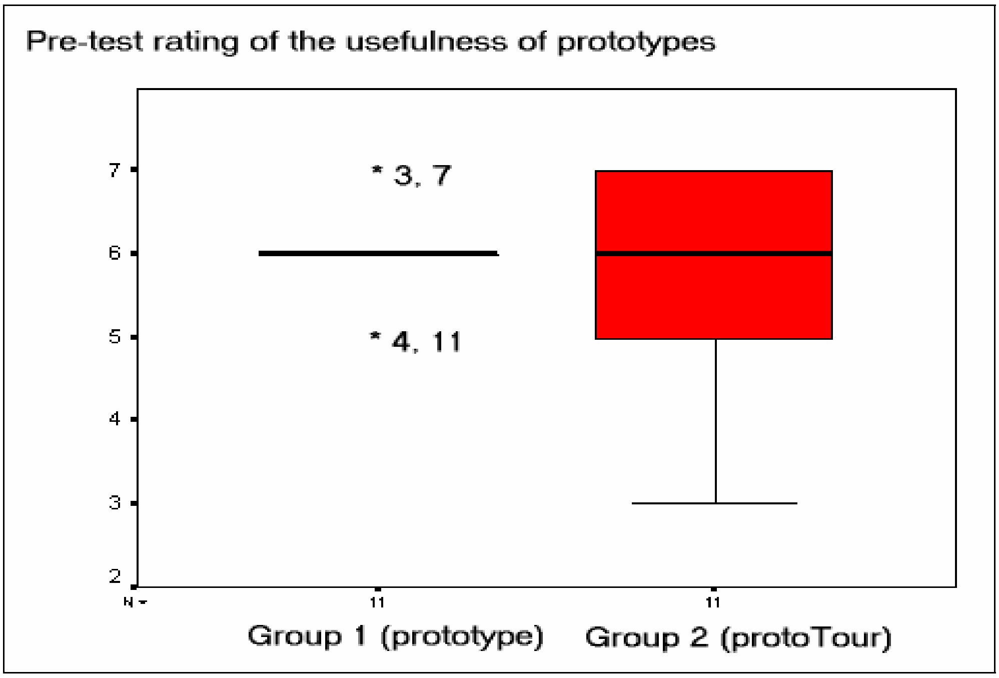


Figure 4.2 Box-plot of the participants' pre-test rating of the usefulness of prototypes

The following table summarises the results of a *t*-test.

Table 4.7 *t*-test of participants' rating of the use of prototypes

	N	**Mean**	**SD**	**SE**	***t***	**df**	**p**
Group 1	11	6.0000	.6325	.191	.392	20	.699
Group 2	11	5.8182	1.4013	.423			

There was no significant difference between groups.

Q1.8 Participants' Rating of the User-Centred Design (or user interface issues) Compared with Functionality

Participants' rated the importance of functionality versus user-centred design within software development on a scale of '1- Functionality is Much More Important than User-Centred Design' to '7 - User-Centred Design is Much More Important Than Functionality'.

Levene's test led to acceptance of the hypothesis that variances are homogenous (F=1.215, p=.284).

The following table summarises the results of a *t*-test.

Table 4.8 *t*-test of participants' rating of the importance of Functionality compared with User-Centred Design

	N	**Mean**	**SD**	**SE**	***t***	**df**	**p**
Group 1	11	4.100	1.101	.348	-.954	19	.352
Group 2	10	4.636	1.433	.432			

There was no significant difference between groups.

Q1.9 Participants' Rating of the Importance of the Role of HCI Designer

Participants' rated the importance of the presence of an HCI designer to the success of a software project on a scale of '1- Unimportant to the Success of the Project' to '7 - Important to the Success of the Project'.

Levene's test led to acceptance of the hypothesis that variances are homogenous (F=1.543, p=.228).

The following table summarises the results of a *t*-test.

Table 4.9 *t*-test of participants' rating of the importance of an HCI designer to the success of a software project

	N	**Mean**	**SD**	**SE**	***t***	**df**	**p**
Group 1	11	6.182	.603	.182	.516	20	.611
Group 2	11	6.000	1.000	.302			

There was no significant difference between groups.

4.4.2 Research Question 1. - Does ProtoTour convey a better understanding of HCI Design intent than a visual prototype?

This section reports the results of experimental measures relating to research question 1.

Experimental Instrument: Expert assessment of participant task performance

The question numbers correspond to the participants' design task and the measure used to assess it. For example, Q3.1b refers to design task 3.1 (Appendix D1.5) and 'b' refers to an expert assessment of this design task (Appendix D sections 1.10 - 1.15).

Q3.1b Expert assessment of the comprehensiveness of the participants' preliminary design

Levene's test for equality of variance led to the rejection of the hypothesis that the variances are homogenous (F=8.165, p=0.010). The following Box-Plot further illustrates this result, showing significantly greater variance in experimental Group 1.

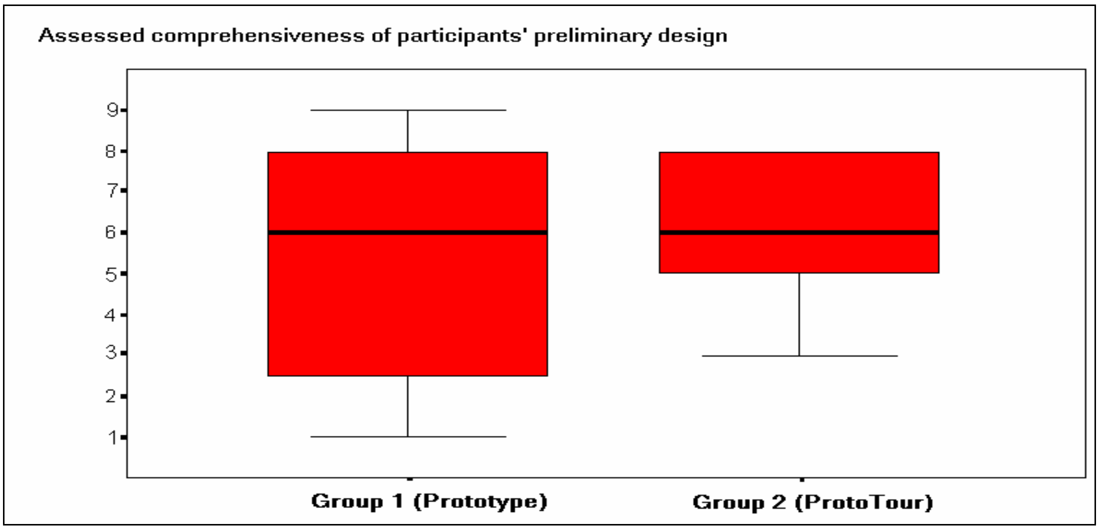


Figure 4.3 Box-plot of the expert's assessment of the comprehensiveness of the participants' preliminary design

The following table summarises the results of a *t*-test.

Table 4.10 *t*-test comparison of 'comprehensiveness' of participants preliminary designs

	N	**Mean**	**SD**	**SE**	**t**	**Df**	**p**
Group 1	11	5.2727	3.101	.935	-.74	20	.466
Group 2	11	6.0909	1.921	.579			

There was no significant difference between groups.

Q3.2 Expert assessment of appropriateness of assumptions made

Levene's test for equality of variance led to acceptance of the hypothesis that variances are homogenous (F=.113, p=.740).

The following table summarises the results of a *t*-test.

Table 4.11 *t*-test comparison of appropriateness of assumptions made by participants

	N	**Mean**	**SD**	**SE**	***t***	**Df**	**p**
Group 1	11	6.7273	2.370	.715	.64	20	.526
Group 2	11	6.0909	2.256	.680			

There was no significant difference between groups.

Q3.3 Expert assessment of appropriateness of implementation difficulties identified

Levene's test for equality of variance led to acceptance of the hypothesis that variances are homogenous (F=2.544, p=.126).

The following table summarises the results of a *t*-test.

Table 4.12 *t*-test comparison of 'appropriateness of implementation difficulties identified' by participants

	N	**Mean**	**SD**	**SE**	**t**	**df**	**p**
Group 1	11	4.9091	2.071	.625	-1.03	20	.314
Group 2	11	6.0000	2.828	.853			

There was no significant difference between groups.

Q3.6a Expert assessment of how well overall, the participants appear to have understood HCI design intent

Levene's test for equality of variance led to the rejection of the hypothesis that the variances are homogenous (F=7.963, p=0.011). The following Box-Plot further illustrates this result, showing significantly greater variance in experimental Group 1. On the y-axis scale, 1 represents 'Very poor understanding of HCI design intent' and 9 represents 'Excellent understanding of HCI design intent'.

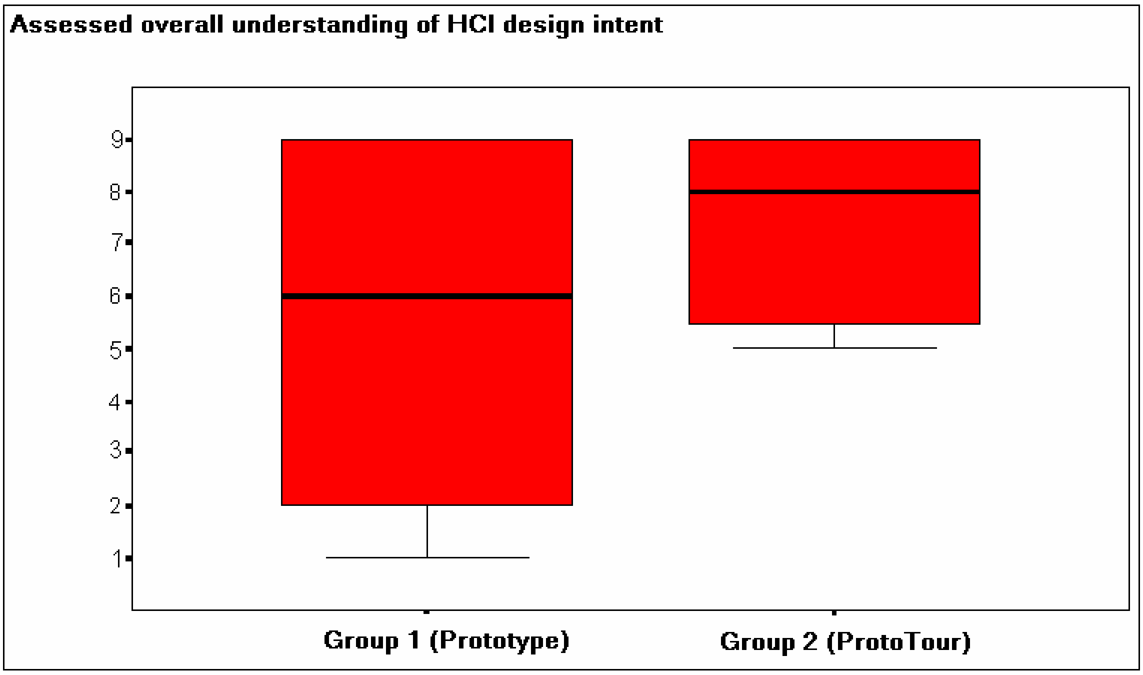


Figure 4.4 Box-plot of the expert's assessment of the participants' overall understanding of HCI design intent

The following table summarises the results of a *t*-test.

Table 4.13 *t*-test comparison of expert assessed 'overall understanding of HCI design intent'

	N	Mean	SD	SE	t	Df	p
Group 1	11	5.6364	3.501	1.055	-1.45	20	.161
Group 2	11	7.3636	1.804	.544			

There was no significant difference between groups.

Q3.6b Expert assessment of consistency of participants' preliminary design with HCI design intent

Levene's test for equality of variance led to the rejection of the hypothesis that the variances are homogenous (F=12.525, p=0.002). The following Box-Plot further illustrates this result, showing significantly greater variance in experimental Group 1. On the y-axis scale, 1 represents 'Completely inconsistent with HCI design intent' and 9 represents 'Completely consistent with HCI design intent'.

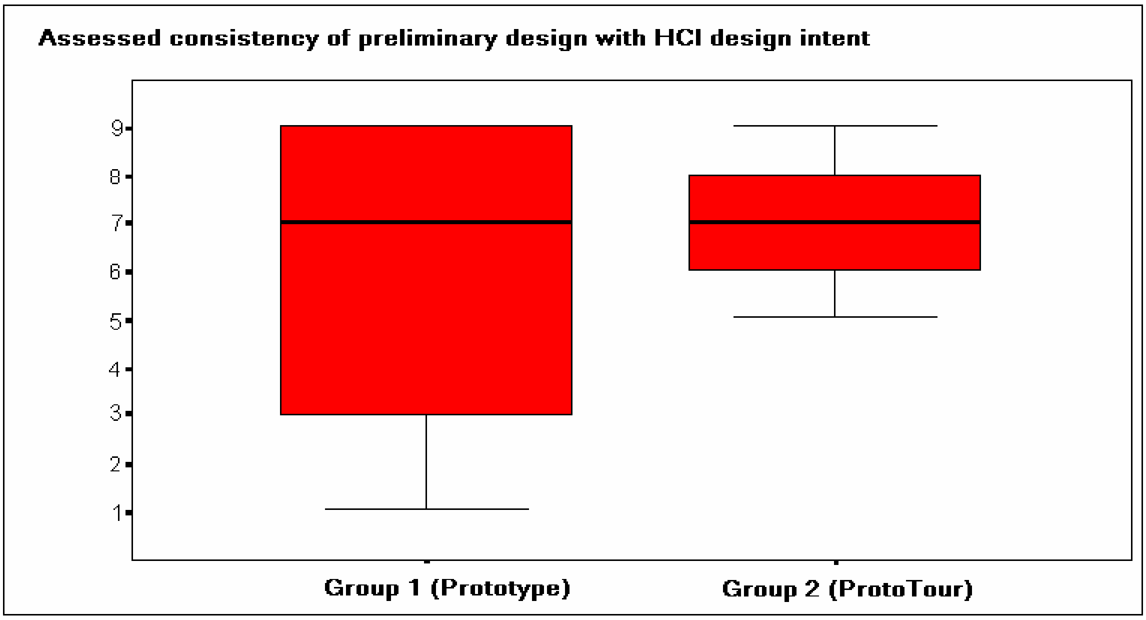


Figure 4.5 Box-plot of the expert's assessment of the consistency of participants' preliminary designs with HCI design intent

The following table summarises the results of a *t*-test.

Table 4.14 *t*-test comparison of 'overall consistency of the preliminary design considerations with HCI design intent'

	N	Mean	SD	SE	t	df	p
Group 1	11	5.8182	3.125	.942	-1.13	20	.271
Group 2	11	7.0000	1.483	.447			

There was no significant difference between groups.

Experimental Instrument: Objective measures of task performance

Q3.5 Participants estimate (in days) of how long the Checklist Window would take to implement

Levene's test for equality of variance led to acceptance of the hypothesis that variances are homogenous (F=.196, p=.663). The following table summarises the results of a *t*-test.

Table 4.15 *t*-test comparison of participants' estimates to implement the Checklist Window

	N	Mean	SD	SE	t	df	p
Group 1	11	7.5455	6.330	1.909	.03	20	.977
Group 2	11	7.4545	7.939	2.394			

There was no significant difference between groups.

A large standard deviation was observed in each experimental group. Further descriptive statistics show that participants' estimates for how long it would take to implement the Checklist Window ranged from 1 to 25 days. Figure 4.6 further illustrates this variability.

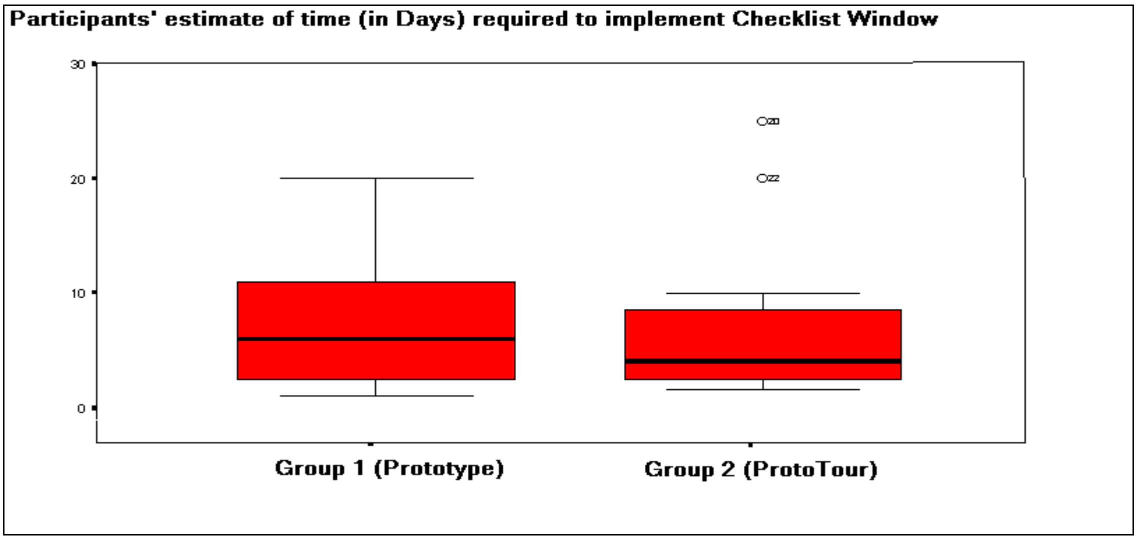


Figure 4.6 Box-plot of participants' time estimates (in days) to implement the Checklist Windows

Note that in the Group 2 Box-plot above, two values coded 'o' are marked. These are values, which are more than 1.5 box-lengths from the 75th percentile are labelled as outliers by SPSS[5] (the value coded 22 represents participant P22, and 20 represents participant P18).

[5] a statistical software package, see Norušis (1993)

Q4.1 Measurement of correctness and completeness of participants' annotation of the functionality of the Checklist Window (post-test)

Levene's test for equality of variance led to acceptance of the hypothesis that variances are homogenous (F=.019, p=0.891).

The following table summarises the results of a *t*-test.

Table 4.16 *t*-test comparison of 'correctness and completeness of participants' annotations of the functionality of the Checklist Window'

	N	Mean	SD	SE	t	Df	p
Group 1	11	4.3182	1.031	.311	-5.40	20	.000
Group 2	11	7.0000	1.285	.387			

Participants of experimental Group 2 (ProtoTour) scored significantly ($p<0.001$) higher than those of Group 1 (prototype).

The variance accounted for (ω^2 estimate) by the participants' utilisation of ProtoTour or prototype was found to be approximately 58%.

The power of the *t*-test with n=11 (per group), with $\omega^2 = 0.5$ at $\alpha=0.01$, is 0.97.

Q4.2 Measurement of correctness of participants' description of the relationship between the checklist and the log file

Levene's test for equality of variance led to acceptance of the hypothesis that variances are homogenous (F=2.456, p=.133).

The following table summarises the results of a *t*-test.

Table 4.17 *t*-test comparison of correctness of participants' descriptions of the relationship between the checklist and the log file

	N	Mean	SD	SE	t	Df	p
Group 1	11	.5455	.270	.081	-1.73	20	.100
Group 2	11	.7727	.344	.104			

There was no significant difference between groups.

Q4.3 Measurement of correctness of participants' description of the relationship between the checklist and the browser

Levene's test for equality of variance led to acceptance of the hypothesis that variances are homogenous (F=.016, p=.901). The following table summarises the results of a *t*-test.

Table 4.18 *t*-test comparison of correctness of participants' description of the relationship between the Checklist Window and the Browser'

	N	**Mean**	**SD**	**SE**	**t**	**Df**	**p**
Group 1	11	.2273	.344	.104	-3.41	20	.003
Group 2	11	.7273	.344	.104			

Participants of experimental Group 2 (ProtoTour) scored significantly (p=0.003) higher than those of Group 1 (prototype).

The variance accounted for (ω^2 estimate) by the participants' utilisation of ProtoTour or prototype was found to be approximately 35%.

The power of the *t*-test with n=11 (per group), with $\omega^2 = 0.3$ at α=0.01, is 0.59.

Q4.4 Measurement of correctness of participants' description of what should happen if advice from a chapter in ELDER is accessed which is different from the current equipment type

Levene's test for equality of variance led to the rejection of the hypothesis that the variances are homogenous (F=39.270, p=0.000). There was significantly greater variance in the correctness of descriptions from experimental Group 1. The following table summarises the results of a *t*-test.

Table 4.19 *t*-test comparison of the correctness of participants' descriptions of what should happen if advice from a chapter is accessed which is different from the current equipment type'

	N	**Mean**	**SD**	**SE**	**t**	**df**	**p**
Group 1	11	.0455	.151	.045	-3.52	20	.002
Group 2	11	.5909	.491	.148			

Participants of experimental Group 2 (ProtoTour) scored significantly (p=0.002) higher than those of Group 1 (prototype).

The variance accounted for (ω^2 estimate) by the participants' utilisation of ProtoTour or prototype was found to be approximately 36%.

The power of the *t*-test with n=11 (per group), with $\omega^2 = 0.3$ at α=0.01, is 0.59.

4.4.3 Research Question 2. - Does ProtoTour improve the accessibility of the animated visual prototype?

This section reports the results of experimental measures relating to research question 2.

Experimental instrument: Participant opinion rating scales and open-ended questions

Q1.7 Pre-test / Q4.10 Post-test rating of how useful participants' perceived prototypes to be

A repeated measures ANOVA was carried out to compare pre and post-test participant ratings of the usefulness of prototypes (results are shown in the following table).

Table 4.20 Repeated Measure ANOVA comparison of pre/post-test participants' ratings of the prototypes' usefulness

Source		Sum of Squares	df`	Mean Square	F	P
Within	Group Error	0 33.636	1 20	0 1.682	0	1.00
Between	Before/After Interaction Error	0 .364 11.636	1 1 20	0 .364 .582	0 .625	1.00 .483

There was no significant change in participants' rating of the utility of prototypes **within** either the prototype or the ProtoTour experimental group. Furthermore, there were no significant differences **between** experimental groups and no significant **interaction** effects.

Q4.5 Participants' rating of how easy it was to visualise the software from the prototype approach adopted

Levene's test for equality of variance led to acceptance of the hypothesis that variances are homogenous (F=2.215, p=.152). The following table summarises the results of a *t*-test.

Table 4.21 *t*-test comparison of 'participants self ratings of how ease it was to visualise the software from the prototype approach adopted'

	N	**Mean**	**SD**	**SE**	**t**	**Df**	**p**
Group 1	11	4.9091	1.758	.530	-1.46	20	.159
Group 2	11	5.8182	1.079	.325			

There was no significant difference between groups.

Q4.6(i) Participants' rating of the utility of supplemental information provided

Data from this measure was not analysed as participants from Group 1 (visual prototype) did not get any supplemental information (although it was available on request) and participants from Group 2 (ProtoTour) were unsure what it referred to. The intention had been to provide Group 1 participants with printouts of the additional information which was available to the other participants within ProtoTour. However, no such information was actually provided during the experiments as queries from Group 1 participants were answered verbally.

Q4.7 Participants' rating of whether they were able to access information about WHY the software was designed as it was

The prototype used by Group 2 (ProtoTour) contained information about design rationale, but the prototype used by Group 1 (visual prototype) did not. Table 4.22 summarises participants' claims of whether they had been able to get information about the rationale for the design.

Table 4.22 Summary of participants' claims whether or not they had been able to get information about the rationale for the design

	Claimed to have been able to get information about Design Rationale	Claimed to have <u>not</u> been able to get information about Design Rationale
Group 1	5	6
Group 2	5	6

There is no difference between the groups of participants' claims.

Q.12 Participants' rating of whether they would like to see prototypes used in a similar way again

Participants were asked whether they would like to see prototypes used in the same way as in the experiment again. These results do not provide a basis for comparison of the prototype and ProtoTour interventions but have value as independent results.

From Group 1, 10 participants were in favour of using that prototyping approach again and one was against.

From Group 2, 10 participants were in favour of using the ProtoTour approach again and one was against.

4.4.4 Research Question 3. - Does the provision of extra information in ProtoTour (including design rationale) enable programmers to offer improved implementation alternatives than if they just had access to a visual prototype and an HCI Designer?

This section reports the results of experimental measures relating to research question 3.

Experimental instrument: Expert assessment of participant task performance

Q3.4a Expert assessment of implementation alternatives identified

Levene's test for equality of variance led to acceptance of the hypothesis that variances are homogenous (F=.247, p=.625). The following table summarises the results of a *t*-test.

Table 4.23 *t*-test comparison of the expert assessment of participants' implementation alternatives identified

	N	**Mean**	**SD**	**SE**	**t**	**df**	**p**
Group 1	11	6.2727	2.533	.764	.09	20	.928
Group 2	11	6.1818	2.089	.630			

There was no significant difference between groups.

Q3.4b Expert assessment of participants' assessments of the likely impact on usability that the implementation alternatives will have

Levene's test for equality of variance led to acceptance of the hypothesis that variances are homogenous (F=.003, p=.956). The following table summarises the results of a *t*-test.

Table 4.24 *t*-test comparison of expert assessment of participants' assessments of the likely impact on usability that the implementation alternatives will have

	N	**Mean**	**SD**	**SE**	***t***	**df**	**p**
Group 1	11	5.3636	3.233	.975	.46	20	.654
Group 2	11	4.7273	3.319	1.001			

There was no significant difference between groups.

Q3.4c Expert assessment of the participants' assessment of the implementation alternatives' likely impact on implementation

Levene's test for equality of variance led to the rejection of the hypothesis that the variances are homogenous (F=4.375, p=0.049). The following Box-Plot illustrates this result, showing significantly greater variance in experimental Group 2.

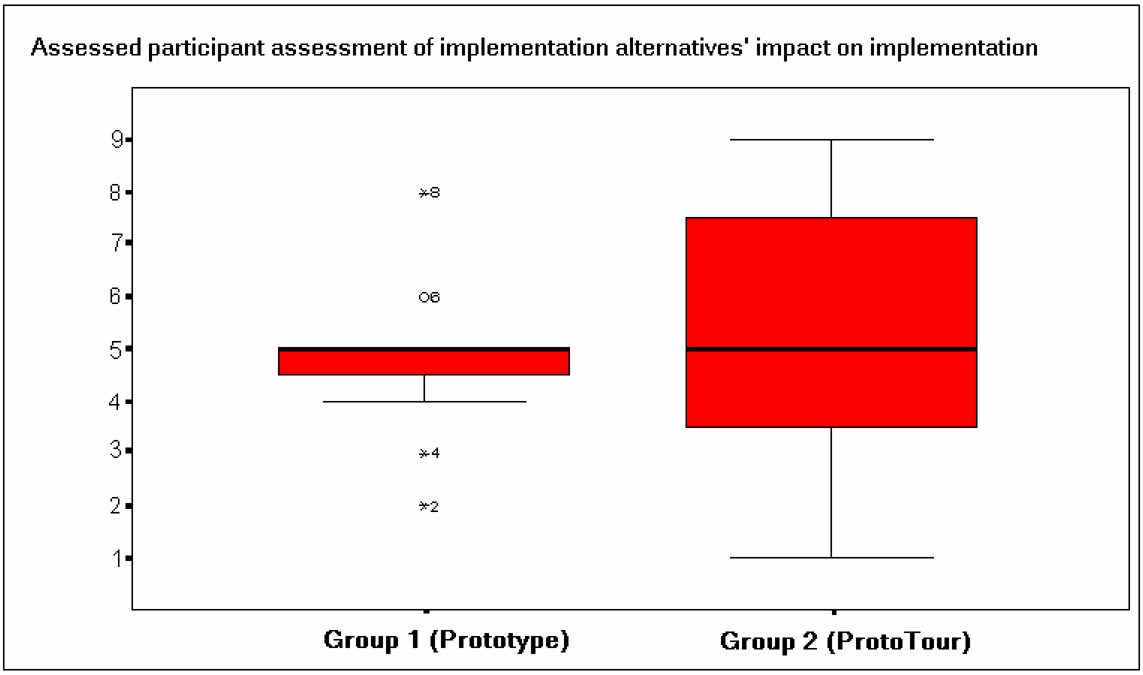


Figure 4.7 Box-plot of the expert's assessments of the participants' assessments of the implementation alternatives' likely impact on implementation

The following table summarises the results of a *t*-test.

Table 4.25 *t*-test comparison of expert assessment of participants' assessments of the implementation alternatives' likely impact on implementation

	N	Mean	SD	SE	*t*	df	p
Group 1	11	4.8182	1.537	.464	-.5	20	.624
Group 2	11	5.2727	2.611	.787			

There was no significant difference between groups.

Experimental Instrument: Objective measures of task performance

Q3.4a(i) Number of implementation alternatives suggested

Levene's test for equality of variance led to acceptance of the hypothesis that variances are homogenous (F=.463, p=.504).

The following table summarises the results of a *t*-test.

Table 4.26 *t*-test comparison of the number of implementation alternatives suggested

	N	**Mean**	**SD**	**SE**	***t***	**df**	**p**
Group 1	11	2.7273	1.902	.574	.36	20	.722
Group 2	11	2.4545	1.635	.493			

There was no significant difference between groups.

Q3.4a(ii) Number of alternatives suggested primarily relating to usability

Levene's test for equality of variance led to acceptance of the hypothesis that variances are homogenous (F=.004, p=.947).

The following table summarises the results of a *t*-test.

Table 4.27 *t*-test comparison of the number of alternatives suggested relating to usability

	N	**Mean**	**SD**	**SE**	***t***	**df**	**p**
Group 1	11	2.4545	2.018	.608	1.25	20	.224
Group 2	11	1.3636	2.063	.622			

There was no significant difference between groups.

Q3.4a(iii) Number of alternatives suggested primarily relating to implementation

Levene's test for equality of variance led to the rejection of the hypothesis that the variances are homogenous (F=17.082, p=0.001). There was significantly greater variance in the number of alternatives primarily relating to implementation from experimental Group 2.

The following table summarises the results of a *t*-test.

Table 4.28 *t*-test comparison of the number of alternatives suggested relating to implementation

	N	**Mean**	**SD**	**SE**	***t***	**df**	**p**
Group 1	11	.3636	.505	.152	-1.94	20	.067
Group 2	11	1.0909	1.136	.343			

There was no significant difference between groups.

Experimental instrument: Participant opinion rating scales and open-ended questions

Q4.8 Participants' rating of how useful design rationale information was in the identification of implementation alternatives

Participants rated the usefulness of design rationale in the identification of implementation alternatives on a scale of 1 'Not useful at all' to 7 'Extremely useful'. There is a large number of missing values in this data set, which prevents statistical analysis. The data are summarised in table 4.29.

Table 4.29 Summary of participants' ratings of the usefulness of design rationale in the identification of implementation alternatives

	Usefulness of Design Rationale in identifying alternative
Group 1	*Value Frequency* Missing 6 Rated 4 4 Rated 6 1
Group 2	*Value Frequency* Missing 6 Rated 1 1 Rated 3 1 Rated 5 2 Rated 6 1

Q4.9 Participants' rating of how useful design rationale was in assessing the impact of alternatives on usability

Participants rated the usefulness of design rationale in the assessment of implementation alternatives' impact on usability on a scale of 1 'Not useful at all' to 7 'Extremely useful'. There is a large number of missing values in this data set, which prevents statistical analysis. The data are summarised in table 4.30.

Table 4.30 Summary of participants' ratings of the usefulness of design rationale in the assessment of implementation alternatives' impact on usability

	Usefulness of Design Rationale in assessing implementation alternatives' effect on usability
Group 1	*Value Frequency* Missing 6 Rated 3 1 Rated 4 2 Rated 5 1 Rated 6 1
Group 2	*Value Frequency* Missing 6 Rated 1 1 Rated 3 1 Rated 5 2 Rated 7 1

4.4.5 Research Question 4. - Does ProtoTour reduce the amount of time that HCI Designers need to spend explaining aspects of the visual prototype to programmers at their request?

This section reports the results of experimental measures relating to research question 4.

Experimental instrument: Observation during the experiment

Obs 3(i) Number of questions participants asked concerning the usability of prototype (Group 1) or ProtoTour (Group 2)

Levene's test for equality of variance led to acceptance of the hypothesis that variances are homogenous (F=.633, p=.435).

The following table summarises the results of a *t*-test.

Table 4.31 *t* -test comparison of the number of questions asked by the participants during the design task regarding the usability of the prototype itself

	N	**Mean**	**SD**	**SE**	***t***	**df**	**p**
Group 1	11	.4545	.688	.207	.64	20	.530
Group 2	11	.2727	.647	.195			

There was no significant difference between groups.

Obs 3(ii) Number of questions participants asked about the intended functionality of the software portrayed in the prototype

Levene's test for equality of variance led to rejection of the hypothesis that variances are homogenous (F=6.251, p=.021). There was significantly greater variance in the number of questions asked by Group 1 (prototype) participants about the intended functionality.

The following table summarises the results of a *t*-test.

Table 4.32 *t*-test comparison of the number of questions asked by the participants during the design task regarding the intended functionality of the software portrayed in the prototype

	N	**Mean**	**SD**	**SE**	***t***	**df**	**p**
Group 1	11	8.0909	6.992	2.108	2.21	20	.039
Group 2	11	2.9091	3.390	1.022			

Participants of experimental Group 2 (ProtoTour) asked significantly (p=0.039) less questions about intended functionality than those of Group 1 (prototype).

The variance accounted for (ω^2 estimate) by the participants' utilisation of ProtoTour or prototype was found to be approximately 16%.

The power of the *t*-test with n=11 (per group), with $\omega^2 = 0.1$ at $\alpha=0.05$, is 0.30.

Obs 3(iii) Number of questions participants asked about the rationale underlying the user interface design presented in the prototype

No questions were asked about design rationale from either experimental group.

Obs 3(iv) Number of questions participants asked about the intended users or their tasks

Only one participant in the study (in Group 1) asked a question in the 'users or their tasks' category.

Obs 3(v) Number of questions participants asked about the scope of the design task set

Levene's test for equality of variance led to acceptance of the hypothesis that variances are homogenous (F=1.261, p=.275).

The following table summarises the results of a *t*-test.

Table 4.33 *t*-test comparison of the number of questions that participants asked about the scope of the design task they had been set

	N	**Mean**	**SD**	**SE**	***t***	**df**	**p**
Group 1	11	1.6364	1.567	.472	1.43	20	.169
Group 2	11	.8182	1.079	.325			

There was no significant difference between groups.

Experimental instrument: Participant opinion rating scales and open-ended questions

Q4.6(iii) Participants' rating of how useful the HCI designer was during the exercise

Levene's test for equality of variance led to acceptance of the hypothesis that variances are homogenous (F=.398, p=.535).

The following table summarises the results of a *t*-test.

Table 4.34 *t*-test comparison of Participants' ratings of how useful the HCI designer was during the exercise

	N	**Mean**	**SD**	**SE**	***t***	**df**	**p**
Group 1	11	5.5455	1.508	.455	1.38	20	.182
Group 2	11	4.5455	1.864	.562			

There was no significant difference between groups.

4.4.6 Qualitative Results - Summary of Participants' Comments and Criticisms

Qualitative data was collected from each participant on the post-test questionnaire.

Experimental instrument: Participant opinion rating scales and open-ended questions

Q4.11 Participants comments and criticisms of the intervention experienced during the experiment as a vehicle for 'hand-over' of the outward software design

Comments and criticisms were broadly categorised (post hoc) to facilitate analysis.

Comments suggesting a need for more information

Table 4.35 contains participants' comments relating to the need for further information to have been supplied in order to carry out the design tasks.

Table 4.35 Participants' comments suggesting a need for more information

Comments from Group 1 (prototype) participants	Comments from Group 2 (ProtoTour) participants
"...there was no information on underlying functionality."	"Some information on amounts of data, sizes of text, etc. were not included."
"Needs supporting documentation of how the HCI designer envisaged parts of the system integrating/effecting each other at time of prototype development..."	"The test suffers from lacking a high level design and the structure in which the exercise fits."
"a visual prototype is not sufficient in itself to fully describe a design."	
"Visual prototypes are very useful in the development process but MUST be accompanied by specification documents in a real development."	

Comments suggesting the need for an HCI designer for further explanation

Table 4.36 contains participants' comments suggesting the need for the presence of an HCI designer to clarify the HCI design intent to accompany the prototype/ProtoTour.

Table 4.36 Comments suggesting the need for an HCI designer for further explanation

Comments from Group 1 (prototype) participants	Comments from Group 2 (ProtoTour) participants
"During the exercise this problem [the prototype gave no information about underlying functionality] was overcome by the HCI Designer providing extra information verbally."	"Animated tutorials, in my view, are an effective sales pitch, but for demonstration of software - even prototype software in pre-development stage the best approach is to have a human demonstrator to provide the immediacy required by the inquisitive receiver of the software."
	"Access to HCI designer still required to clarify some points."
	"I wouldn't solely want the ProtoTour as an illustration, but combined with liaison with the HCI designer looks pretty good!"

Comments expressing concern about the problems caused by users' and clients' increasing expectations when using a prototype approach

Table 4.37 contains participants' comments expressing concern about the problems caused by users' and clients' increasing expectations when using a prototype approach.

Table 4.37 Comments expressing concern for the expectations which prototypes can cause

Comments from Group 1 (prototype) participants	Comments from Group 2 (ProtoTour) participants
"... it may be difficult for the user to understand that a prototype has no functionality."	
"Criticisms. May raise user expectations."	
"CRITICISMS: If potential users have seen it [the prototype] demo'ed their expectations be different from what is finally implemented."	
"(1) Sometimes visual prototyping can mislead the customer into thinking they already have a close to working system when most of the functionality does not in actual fact exist."	

Comments highlighting the need for the programmer and HCI designer to work together

Comments from a number of participants suggest that programmers and HCI designers should work together. These are listed in table 4.38.

Table 4.38 Comments highlighting the need for the programmer and HCI designer to work together

Comments from Group 1 (prototype) participants	Comments from Group 2 (ProtoTour) participants
"Coder should be included in the prototyping stage - so can point out some implementation issues - i.e., if something will be impossible to implement - or will take too much time - computer resources, etc."	"A general criticism would be that designers and programmers should always work together to ensure "realistic" demos."
"Criticisms. – Functionality may be difficult to implement."	
"CRITICISMS: HCI Designer might not understand what is required to actually implement the design."	

Comments expressing concern about cost and effort required to construct prototype/ProtoTour

Some participants expressed concern about the cost and effort which would be required to construct ProtoTour. These are reproduced in table 4.39.

Table 4.39 Comments expressing concern about cost and effort required to construct prototype/ProtoTour

Comments from Group 1 (prototype) participants	Comments from Group 2 (ProtoTour) participants
	"Would be concerned about the cost of the exercise – why doesn't the designer code the display himself? – probably cheaper and it will be as he wants it first time."
	"I have doubts about the effort it must take to produce a ProtoTour..."
	"who would do additional work to annotate and explain the prototype in the correct format."

Comments concerning usability of prototype/ProtoTour

Several participants' comments relate to usability issues surrounding the prototype/ProtoTour. These are listed in table 4.40.

Table 4.40 Comments concerning usability issues with prototype/ProtoTour

Comments from Group 1 (prototype) participants	Comments from Group 2 (ProtoTour) participants
"I wasn't completely clear how the Checklist was supposed to work..."	"Vague as to where I was in the ProtoTour illustration and how much of it I had viewed, probably because I was on a tight time scale. Information should be kept to as much as can be fitted on a screen, not so much scrolling. A few more pop-up definitions would have been good rather than jumps to other topics."
	"Animated tutorials can take full control away from the user and can be frustrating in the sense that the machine becomes the 'know it all' and the user is left dumb. It is more satisfying for the user to find answers to his questions himself rather than have them laid on a plate. That said, a good animated walkthrough gives an excellent and concise overview or 'feel' of what the software is or isn't going to be."
	"Far better than (eg) a VB [VISUAL BASIC] prototype to play with as there was more information."
	"In the timed exercise I felt there was too much information available and I felt a bit swamped..."

Comments suggesting that prototypes are useful for getting requirements right

Some participants highlighted the usefulness of prototypes for getting the proposed software's requirements right, these are reproduced in table 4.41.

Table 4.41 Comments suggesting that prototypes are useful for getting requirements right

Comments from Group 1 (prototype) participants	Comments from Group 2 (ProtoTour) participants
"Prototype helps to capture user requirements accurately..."	"I think the main use of these prototypes is to investigate how the user sees the problem. "
"A prototype is only really useful if the 'actual' users of the end system use the prototype as they must use the system in the end."	
"A usefulness of prototyping is that, if done properly, it can indicate where future problems shall arise."	

Other positive comments about the prototype or ProtoTour

Other positive comments participants made about the prototype or ProtoTour are reproduced in table 4.42.

Table 4.42 Other positive comments made about the prototype or ProtoTour

Comments from Group 1 (prototype) participants	Comments from Group 2 (ProtoTour) participants
"The visual prototype was good for showing the appearance and behaviour of the interface..."	"...a good animated walkthrough gives an excellent and concise overview or 'feel' of what the software is or isn't going to be."
"Visual prototypes are extremely useful. The design process should be iterative and it is expected that many alternatives are explored before arriving at a suitable design."	"It quickly gave me a view of what was required."
"...prototyping can give the coder a better understanding of how the overall system should look - a kind of this is what we are aiming for."	"Yeah, its good... ...In the timed exercise I felt there was too much information available and I felt a bit swamped but in a real situation this information would prove invaluable. It serves to support the HCI Designer's view and gives the implementor confidence that the HCI Designer has thought through the issues with rigour. "
"Useful as it already defines forms, appearances and interactions. Aids design of functionality as it is known what is required of each form"	"Looks good. Far better than (eg) a VB [VISUAL BASIC] prototype to play with as there was more information."
"Visual prototypes are very useful in the development process... ...A 'dynamic' prototype showing the system may be helpful, this could be a walkthrough as in the exercise and/or annotated screenshots in a (powerpoint?) presentation"	"ProtoTour illustration is useful..."

Other specific comments of interest

Other comments made by participants, which provide various insights, are reproduced in table 4.43.

Table 4.43 Other specific comments of interest providing various insights

Comments from Group 1 (prototype) participants	Comments from Group 2 (ProtoTour) participants
"Major criticism is that prototypes can guide implementation too much – coder can lose sight of some implementation issues and may be forced to produce something that could have been implemented in a better way."	"An understanding of WHY is VERY important in making assessments of alternatives - design rationale is often not recorded."
"For simple projects its probably OTT."	"In general for complex applications the structure of the interface should not be the main source of information guiding the underlying designs, as potentially these may have to work with changing or radically different interfaces."
"Care must be taken not to sacrifice functionality for ease of use."	
"A 'dynamic' prototype showing the system may be helpful, this could be a walkthrough as in the exercise and/or annotated screenshots in a (powerpoint?) presentation"	
"I wasn't completely clear how the checklist was supposed to work, not all of the Notes entered were displayed in the log file – maybe this is a bug."	

4.5 Discussion

An observation relating to the characteristics of the sample is described first. Discussion of results relating to each research question follows; reported by experimental instrument, and each having a separate conclusion. Findings additional to the main research questions are then discussed.

4.5.1 Characteristics of the Sample

A noteworthy observation relating to the characteristics of the sample is evident from comparing tables 4.2 and 4.3. Twelve programmers (55%) claimed to have been programming for more than five years (table 4.2). However, when asked how many years that they had been programming user interfaces, only seven programmers (32%) reported more than five years (table 4.3). There are at least three possible explanations for this discrepancy. Firstly, some programmers may have spent the majority of their programming career solely concerned with deep internal code that did not interact with any user. Secondly, the advent of GUIs has greatly increased the amount of code that the user interface now requires (as described in section 1.2.2), so programmers are now more likely to work on an assignment which involves user interface programming. A third explanation is that some programmers perceive user interface programming to have started with the advent of GUIs and Windowing environments. Some programmers apparently do not understand that a user interface is any aspect of the computer that affords interaction with the user. Thus, there appears to be a need to further educate programmers about the meaning of HCI design and usability.

Familiarity with Microsoft Windows Help was used as a secondary consideration in the assignment of individuals to experimental groups. However, a 1-tailed *t*-test showed there to be no significant difference in the mean participant ratings of their familiarity with Windows Help between the experimental groups.

4.5.2 Discussion of Research Question 1

Research Question 1 - Does ProtoTour convey a better understanding of HCI Design intent than a visual prototype?

This research question was addressed using two kinds of experimental instrument: expert assessments of task performance; and objective measure of task performance.

Experimental instrument: Expert assessment of participant task performance

There were no significant differences in the mean participant performance between the experimental groups on any of the expert assessed measures. The remainder of this section suggests possible explanations.

It is likely that the slightly artificial nature of the programmers' design task reduced the potential for discrimination between the experimental groups. In the original design task, programmers were asked to produce a preliminary design of the proposed ELDER software application shown by the interventions. The intention was for programmers to produce a coarse preliminary design covering the whole system (in fact, the ProtoTour intervention constructed does indeed describe the entire ELDER application). This would have been much closer to a realistic programming task than the final design task, which involved carrying out the preliminary design of the Checklist Window of ELDER alone. The original design task was changed after the pilot study; it was perceived to be too much work for programmers, within the tight time constraints of the experiment. Producing a preliminary design for the Checklist Window was a more constrained problem than programmers would usually meet. This gave them less scope for introducing new ideas or for suggesting how the implementation might be improved. When looking at a proposed system as a whole, programmers are likely to suggest the removal or re-design of chunks of the application. This might involve questioning why certain Windows are required, or suggesting alternative windows to those shown in the HCI design. With the Checklist Window, the participants did not get an opportunity to express such creativity. They were focused on just one Window and were not concerned with how this fitted in with the rest of the system. Looking at the application as a whole, programmers may well have questioned the need for the Checklist Window. Just looking at the Checklist Window in isolation did not give them this broad view.

The experiment also failed to model 'real world' commercial programming practice in another way. In the 'real world', a programmer may begin programming the Checklist Window with only a rough mental model of how it should work. As the artifact evolves through the implementation, it would change in ways that programmers may not have predicted or may not have remembered to specify in preliminary design work. For example, the first version of the Checklist Window may have no means of closing down. Although this may not be at the forefront of the programmer's mind during the preliminary design, when they first run the Checklist Window they will realise that it cannot be closed. After adding a close mechanism, the artifact (program) itself will prompt the programmer to consider what other events should occur when the Window is closed. Thus, the evolving artifact itself becomes a prompt to the programmer. However, it is because of the limitations of this prompt that misunderstandings and omissions occur. Within the experiment, because programmers were designing on paper, misunderstandings that arise from these 'prompts' were avoided. This reduced the experiment's potential for assessing misunderstandings between experimental groups. In practice, this kind of prompt may create a decision point for programmers. For example, they would have to either find out what should happen when the Checklist Window closes or make an assumption. This is the point at which ProtoTour could provide useful support for such decisions based on HCI design intent. If ProtoTour was accessible to programmers, it may encourage them to seek an answer in line with HCI design intent, rather than to make an inappropriate (or, ill-advised) assumption.

Expert assessment of participants' performance during this experiment was quite coarse because of time constraints. The apparent lack of significant differences in mean group performance could be explained by this. Only serious misunderstandings would have been evident in this coarse assessment. Thus, the design tasks completed by the participants provided less information for the experts to assess than if they had actually implemented their preliminary designs. If participants had been able to implement their preliminary designs, the expert would have had more to work on.

A further contributor to a lack of significant results is likely to be the severe individual differences among commercial programmers that are known to exist (Curtis, 1988; Sheil, 1987; Weinberg, 1971; McGarry, 1982). This is a recognised confounding variable in experiments involving programmers (Curtis, 1981). One illustration of individual differences can be observed by analysing the design representations that participants adopted. During the expert assessment, the style of representation and/or notation used by each participant was categorised. Seven different categories of representations were employed by the participants and several participants used more than one kind. Representations comprised textual descriptions (x6.5); Object-Oriented representations (x5.5); Flow Charts (x4.5); Pseudo Code (x2.5); State Transition Diagrams (x1.5); Header File (x1); Data Flow Diagram (x.5). As the majority of the participants were programmers of Windows applications involving object-oriented concepts, it is surprising that such a diversity of representations was used in the study. This seems to provide an indication that, there are distinct differences in programmers' preferred design representations if they are given freedom to choose an appropriate design representation. Thus, individual differences are apparent in the way programmers go about solving their normal work tasks. This supports the findings of Curtis et al (1987).

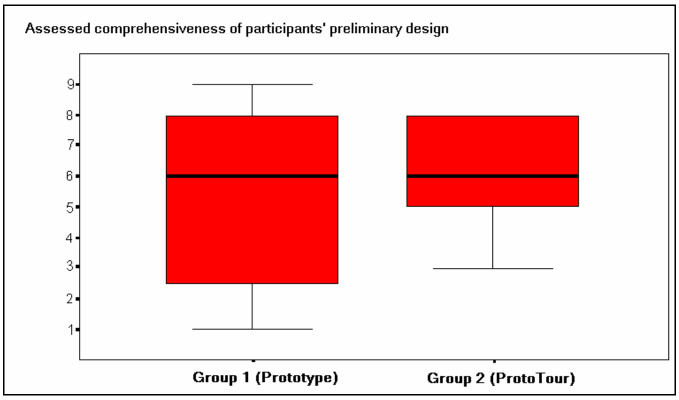
The relatively small sample size may have also contributed to the failure of expert assessed measures to detect significant results. However, this is a feature of studies involving commercial programmers, because their time is so expensive. Many studies involving programmers actually use programming students as participants; a practice which is regarded as highly questionable (by Brooks, 1980).

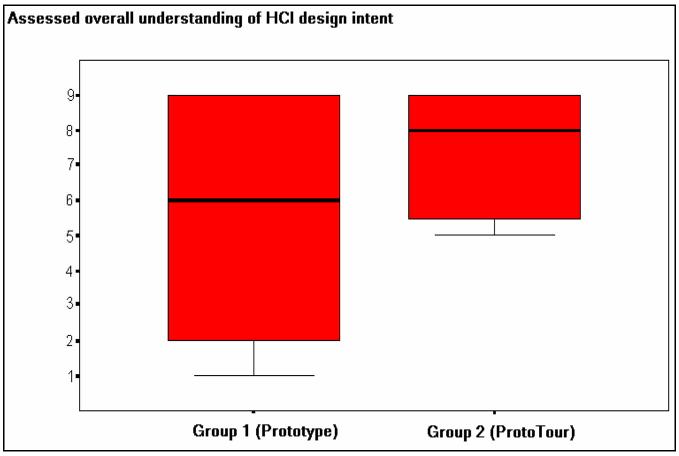
Although there were no significant mean performance differences found by the expert assessed measures relating to research question 1, **several measures did show significant differences in the variance of participants' performance between experimental groups**.

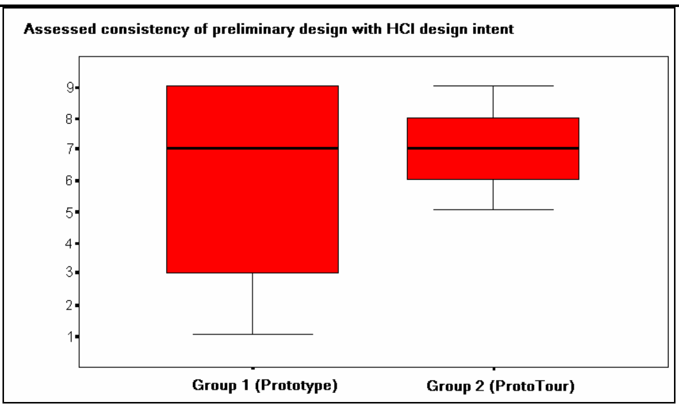
Three performance measures, showed significant differences in the variance between the two groups (see table 4.44). In each case, there was less variance in the performance measures from the Group 2 (ProtoTour).

Table 4.44 Significant differences in the variance of expert assessed measures relating to research question 1

Measure	**Levene's test for equality of variances**	**Box-plot**

Expert assessment of the comprehensivenes s of the participants' preliminary design (Q3.1b)	F=8.165
	p= .010
		

Expert assessment of how well overall, the participant appears to have understood HCI design intent (Q3.6a)	F=7.963
	p=.011
		

Expert assessment of consistency of participants' preliminary design with HCI design intent (Q3.6b)	F=12.525
	p=.002
		

These disparities in variance provide an indication that the ProtoTour intervention reduced the variability in the programmers' performance. Thus, the comprehensiveness of participants' preliminary designs, their apparent understanding of HCI design intent and the consistency of their designs with HCI design intent were all **significantly less**

variable for Group2 (ProtoTour) than for Group 1 (prototype) participants. The quartile ranges shown in the above box-plots show that this effect is not caused solely by individual outliers.

This result provides an indication that ProtoTour has proved to be beneficial in raising all Group 2 programmers' understanding of HCI design intent. The information and explanation it provided appears to have benefited all participants of this group. None of the programmers in Group 2 performed badly, each seeming to have understood HCI design intent at some level. For example, in table 4.44, Q3.6a has a low rating of 1 for Group 1 (prototype) and 5 for Group 2 (ProtoTour). The lower quartile is 2 for Group 1 and 5.5 for Group 2. Although ProtoTour has not raised the peaks of programmer performance (and there are no significant differences between mean performance on these measures), it does seem to have ironed out the troughs to some extent.

It is possible that the ProtoTour intervention does not greatly assist the more skilled programmers. Perhaps highly skilled programmers are able to derive enough information about HCI design intent from the visual prototype alone when they focus on one window in isolation. It is unlikely that this would apply for a large project where a programmer had to consider the HCI design intent of the entire system. However, when designing an individual Window, the added value of using ProtoTour for highly skilled programmers may be low. As has been argued from the literature review (section 1.10.2.1) and can be seen from the qualitative investigation described in chapter 2, programming teams are usually comprised of mixed ability programmers. The reduction in variability of the programmers' assessed performance is perhaps an indication that ProtoTour helped to keep the less skilled programmers on-track. For a large software project, this may well be a benefit worth having. Misunderstandings in a software project can be very costly (Mantei and Teorey, 1988).

If ProtoTour truly could reduce the variability in programmers' performance during an implementation task, this would be a very strong case for its adoption in mixed ability software teams.

Using expert assessed measures, the ultimate test ProtoTour's utility for conveying HCI design intent would be to examine programs built by participants of two experimental groups, using the ProtoTour and prototype interventions. The assessor could then more effectively concentrate on the end result of a programmers' understanding of HCI design intent.

Experimental Instrument: Objective measures of task performance

In this area there were highly significant results to indicate that participants from the Group 2 (ProtoTour) gained a better understanding of the HCI design intent and functionality of the Checklist Window, than Group 1 (visual prototype) participants. However, one objective measure proved less effective.

The ineffective measure concerned participants' estimates of how long it would take them to implement (program) the Checklist Window (Q3.5). It was believed that members of one experimental group might see the implementation problem as more involved than members of the other group, if they better understood the HCI design intent. However, in the experiment, estimation turned out to be an ineffective measure. There was large variability in the estimates from participants of both experimental groups, and there were no significant differences between the mean estimates of each group.

One possible explanation is that the *t*-test did not account for the programming languages on which programmers had based their estimates. However, the design task did allow programmers to select a development language, so it could be expected that they would estimate for the most familiar language. It is a strong possibility that severe variability in estimates would have been found whether or not the programming language was factored out. Estimation in commercial programming is notoriously difficult (McCarthy, 1995), and programmers are said to be particularly bad at it (e.g. by Brooks, 1975 and Stephens, 1996). In fact, this experiment supports these previous findings. Across the experiment, estimates varied from 1 to 25 days with a mean of 7.5 days and a standard deviation of 7.0 days. Considering that the sample comprised 22 commercial programmers and the Checklist Window exhibited only modest complexity, it would seem reasonable to expect less variability. One possible explanation for the variation is that programmers were asked to incorporate some of their implementation alternatives into the estimate. However, revisiting these in some detail has shown this to be an unlikely explanation for the variability. It is widely recognised that individual differences exist in various performance measures used to examine programmers; some studies report difference ratios of 20:1 (see section 1.10.2.1) and other studies have shown similar orders of magnitude differences. Although a fundamental aspect of a programmer's job, estimating is probably not a particularly good measure of programmer performance. However, if it were considered such, the order of magnitude differences in these estimates is consistent with differences found in other performance measures. This dimension perhaps illustrates the individual differences which existed within the sample.

A further insight can be gained from the pilot study. In the first pilot study, participants were reluctant to make any estimates at all (see section 4.3.7.1). Perhaps this reluctance to estimate and the variability of the estimates made during the experiment shows that programmers realise estimation is a problematic task. This possibly leads some to providing estimates as a guide only (not taking it very seriously), whilst the others make high estimates to cover themselves.

Other objective measures of task performance relating to research question 1 did provide significant results. As part of the post-test, participants completed the annotation of the Checklist Window they had been working on (Q4.1). The participants of Group 2 (ProtoTour) produced significantly ($P<0.001$) more correct and complete annotations than Group 1 (visual prototype). Annotations were marked on a pre-determined 10 point marking scheme. The strength of association approximation suggested that an extremely high 58% of the variance was accounted for in this test. The power of the test was also estimated to be extremely high at 0.97. **This is perhaps the most important finding of the experiment**, partly because of the measure's objective nature and partly because of the level of significance of the result.

Similar post-test questions asked participants to state the relationship of the Checklist Window to other parts of the proposed ELDER software. On two of these questions, the performance of Group 2 participants (ProtoTour) was significantly better. There were significantly more correct descriptions of the relationship between the Checklist and the Browser (post-test - Q4.3) from Group 2 participants ($p=0.003$; $\omega^2 = 0.35$; power = 0.59). Similarly, there were significantly more correct descriptions of what should happen if advice from a chapter is accessed which is different from the current equipment type (post-test - Q4.4) ($p=0.002$; $\omega^2 = 0.36$; power = 0.59).

All of these comprehension-style post-test questions were intended to assess whether the participants had understood HCI design intent. The Group 2 participants' significantly better performance on these measures is thought to be because ProtoTour does indeed help to convey a better understanding of HCI design intent than a visual prototype.

Conclusion

Direct questioning of participants' **understanding** of how the Checklist Window should work, indicated that the Group 2 (ProtoTour) participants had gained a significantly better understanding of HCI design intent than Group 1 participants. However, the experiment did not appear to provide a good mechanism for assessing their performance **based** on this understanding.

On three separate objective measures, when questioned about their understanding of the Checklist Window and how this fitted with other aspects of the ELDER system, the Group 2 (ProtoTour) participants gave more correct responses than Group 1 participants (significance of $P<0.005$ for all three measures). Consequently, it is believed that the ProtoTour intervention is capable of conveying a better understanding of HCI design intent than a visual prototype, supported by an HCI designer.

Expert assessed measures of the participants' work based on their understanding did not discover any significant differences in mean performance between the groups. Although there are several explanations for this, there is probably one main reason: the tasks were not sufficiently realistic. In commercial practice, a programmer would have considerably more time to carry out the preliminary design work, and in all likelihood, they would have a larger part of the implementation to carry out. Although expert assessed measures are believed to be invaluable, future experiments should strive to provide experts with the actual programs that participants implemented based on the HCI design intent shown in either ProtoTour or the visual prototype.

Expert assessed measures did, however, provide some indication of ProtoTour's potential. On three different measures, the task performance of participants in Group 2 showed significantly less variance than that of Group 1. In all of these cases, it seemed that the ProtoTour intervention had 'lifted' the performance of the lower performing participants. Although this is a weak result in the context of this experiment, it is worth future research. If ProtoTour does nothing much for the 'best' programmers but raises the performance of the 'worst' programmers, it could well have value in a mixed ability team (especially as most programming teams are thought to be mixed ability – see section 1.10.2.1).

Although not directly relevant to the research question, the variability of estimates provides some support for previous findings in individual differences among programmers, and the difficulties of estimating implementation duration in software.

4.5.3 Discussion of Research Question 2

Research Question 2. - Does ProtoTour improve the accessibility of the animated visual prototype?

This research question was addressed a single type of experimental instrument based on participant opinion scales and open-ended questions. The measures introduced to address this research question could provide only an indication of the accessibility of the interventions. The rationale for these items was to provide an insight into whether programmers would consider the interventions useful. A strong positive result from programmers on these tangential measures would have provided evidence that they would accept the intervention into their work, thus indicating the early stages of accessibility.

Experimental instrument: Participant opinion rating scales and open-ended questions

Participant opinion measurements were used to get an idea of whether participants favoured the ProtoTour representation over the conventional visual prototype. Overall, there were no statistically significant results to indicate that ProtoTour would be more accessible than a conventional visual prototype. The remainder of this section suggests possible explanations for this.

Of the measures devised for research question 2, a pre-test (Q1.7)/post-test (Q4.10) rating of the prototypes' utility was expected to have the greatest potential for discriminating between the experimental groups. A repeated measures ANOVA was used to analyse within and between groups effects. No significant results were found in participants' utility ratings of prototypes before and after the experiment, in either group[6]. Nor were there significant differences between the groups. This lack of significant results could perhaps be attributed to a design flaw in the questionnaire item, which was not identified in either of the pilot studies. The scale used to rate visual prototypes was a seven point Likert-type scale with 1 representing 'Not useful at all' and 7 as 'Extremely useful' (See Appendix D1.3 and D1.6). Responses to the items were mainly bunched-up at the top end of the scale (i.e. on the pre-test seven participants rated '7' and nine participants rated '6') causing a ceiling effect. This lack of spread of responses confounded analysis and any possible discrimination between the experimental groups.

[6] Group 2 participants were expected to regard ProtoTour as a type of prototype

There was no difference found in the ratings ascribed to how easy it was to visualise the ELDER software from using either ProtoTour or the visual prototype (Q4.5). This was perhaps because the experiment design only exposed participants to one intervention. However, it is hard to envisage an experiment in which participants could assess the two interventions and subsequently provide a useful rating of how easy it had been to visualise the software from each. With the benefit of hindsight, this measure would appear to be optimistic.

Asked whether they were able to get information about why the Checklist Window was designed as it was (Q4.7; table 4.22), five participants from each experimental group claimed they were and six claimed they were not. Therefore, there was no apparent difference between the accessibility of design rationale information to the two experimental groups. However, ProtoTour contained design rationale information within it but the visual prototype did not. Participants using the visual prototype had only one route to design rationale information and that was the HCI designer (the experimenter) – this was done to simulate usual commercial practice. No participants asked for or received any information about design rationale from the experimenter (Obs. 3(iii); Appendix D1.5) and none was ventured unprompted by the HCI designer. Therefore, the five participants from the visual prototype group who thought they had been able to gain design rationale information, were mistaken.

One explanation for this could be that this post-test item asked participants to reflect on the experiment. Although no participants asked questions about design rationale, most did ask some questions about the **design intent** of the Checklist Window. It is possible that participants' felt that these were questions about design rationale. Because five participants from Group 1 were mistaken in this way, doubt is cast on the results from Group 2, where five participants also claimed to have gained information about design rationale. However, participants of Group 2 did have access to design rationale through ProtoTour, although it is not possible to be certain whether they used it. There is one indication that design rationale information within ProtoTour was accessible from the comments of one participant in Group 2:

"An understanding of WHY is VERY important in making assessments of alternatives - design rationale is often not recorded." (table 4.43)

There is some debate in HCI about the benefit of capturing design rationale. Is the benefit of recording such information worth its cost in terms of collection and representation effort? (Buckingham-Shum, 1996; Grudin, 1996). Such doubt illustrates that it is not common practice in software engineering to state the underlying rationale. This experiment perhaps supports this supposition. None of the programmers in the experiment questioned the rationale for the design; five participants who did not receive such information felt that they had. This perhaps demonstrates that programmers are not used to receiving such information or being in a position to question the rationale.

An assessment of the general accessibility of using a prototype-centred approach (which includes both ProtoTour and the visual prototype) was included in the post-test (Q1.12). This asked participants if they would be in favour of using prototypes in a similar way in future. The measure was not designed to afford a comparison between the experimental groups, rather it aimed to gain an overall impression of programmer opinions to the general prototype-centred approach. It should be recognised that, strictly speaking, the participants in each experimental group were rating something different. However, this interpretation of the results concerns the general category of prototype-centred explanation, which covers both interventions.

In each experimental group, 10 out of the 11 programmers (91%) said they would like to see the same intervention used in a similar way on a future software development. Programmers showed a preference for using an intervention similar to prototype or ProtoTour, compared with NOT using this sort of intervention. Furthermore, five participants in each experimental group made positive comments about the approach they had used in the experiment (Q4.11; table 4.42). This provides further evidence of programmer support for prototype-centred explanation which is likely to contribute favourably to the accessibility of prototype-centred explanation.

Other issues surrounding accessibility were evident from participants' comments and from the observations of the experimenter. During the experiment, some participants from Group 1 were seen to explore the limits of the visual prototype, as if it were an accurate literal representation of ELDER. In fact, some comments illustrate that they had done this, for example:

> "I wasn't completely clear how the Checklist was supposed to work, not all of the Notes entered were displayed in the log file - maybe this is a bug." (from table 4.40)

The participant's reference to a potential bug in the visual prototype illustrates a lack of understanding of what the visual prototype is. This misunderstanding could lead to programmers mistrusting the representation shown in the prototype and therefore refrain from referring to it. This would clearly have a negative impact on accessibility of visual prototypes.

However, a similarly important comment was made by a participant of the ProtoTour group:

> "Animated tutorials can take full control away from the user and can be frustrating in the sense that the machine becomes the 'know it all' and the user is left dumb. It is more satisfying for the user to find answers to his questions himself rather than have them laid on a plate. That said, a good animated walkthrough gives an excellent and concise overview or 'feel' of what the software is or isn't going to be." (From table 4.40)

This participant is concerned that the animated tutorials take away control from the user. Leaving the user in control is a recognised heuristic in HCI and this participant feels it has been violated by the automated walkthroughs. Were this sentiment to be felt more widely, it could have a negative impact on the accessibility of ProtoTour.

Conclusion

These results provide no statistically significant evidence that ProtoTour is more accessible than a conventional prototype. However, there were indications that prototype-centred explanation is an approach favoured by programmers. Findings relating to design rationale are also of interest.

The prototype-centred approach to conveying design intent was well received by programmers in both experimental groups. In both groups, 91% of programmers said they would like to see the respective intervention used on future software projects. Although this does not provide a comparison between the two prototype-centred interventions used in the experiment, it does indicate that programmers are generally in favour of such approaches. This claim is further supported by positive comments about each intervention from five of the eleven participants in each group. This favourable rating of the use of prototype-centred explanation provides a positive indication that programmers would find such approaches accessible.

Although not directly relevant to the research question, findings relating to design rationale are of interest. It is interesting that five programmers believed they had received information relating to design rationale when they had not. The discussion suggests that it is perhaps because programmers are unused to receiving such information that they confuse it with information about design intent. Thus, this item begins to confirm that design rationale is not routinely used within software development (see section 1.11.3.4).

Finally, comments from participants did indicate some possible negative indications concerning accessibility. A participant of the Group 1 misunderstood the nature of the prototype, complaining about a possible 'bug'. The potential for such misunderstandings is part of the motivation for the creation of ProtoTour. However, one ProtoTour participant did not like the automated walkthroughs because it took control away from the user. These findings perhaps demonstrate that not all aspects of ProtoTour would be welcomed by programmers. However, automated walkthroughs would prevent the misunderstandings about bugs in the prototype, but this would be at the expense of the user being in control.

It is likely that the question of accessibility would also be better addressed in a more comprehensive and large scale experiment using a real software project. A participant-observation approach to such a study would allow an assessment of how often reference was made to the prototype/ProtoTour during the project. This would possibly provide a better indication of the accessibility of the two interventions, although rigorous comparison would be problematic.

4.5.4 Discussion of Research Question 3

Research Question 3. - Does the provision of extra information in ProtoTour (including design rationale) enable programmers to offer improved implementation alternatives than if they just had access to a visual prototype and an HCI Designer?

Three experimental instruments were designed to assess this research question: expert assessment of task performance; objective measures of task performance; and participant opinion rating scales and open-ended questions.

Experimental instrument: Expert assessment of participant task performance

There were no significant differences in the mean participant performance on any of the expert assessed measures between the experimental groups. Some of the other expert assessed measures used in the experiment also failed to discriminate between experimental groups. Thus, the absence of significant differences on these measures is likely to be for similar reasons (see section 4.5.2).

To summarise, expert assessed measures failed to detect significant differences in mean participant performance between experimental groups for items 3.4a, b and c (Appendix D1.13-1.14). The first measure assessed the quality of the participants' implementation alternatives, and the second and third measures assessed how well the participants had understood the likely impact of the alternatives suggested.

It can be considered a positive result for ProtoTour that there were no significant differences in the quality of implementation alternatives suggested by participants of the ProtoTour and the visual prototype groups. During the experiment, ProtoTour participants asked the HCI designer significantly fewer questions about HCI design intent (see results relating to research question four) and yet suggested implementation alternatives that were as good as those proposed by the visual prototype participants.

Experimental Instrument: Objective measures of task performance

There were no significant differences in the objective measures of task performance between the experimental groups.

These measures consisted of counting the number of implementation alternatives proposed, and counting how many of these related to usability, and how many related to implementation.

These measures show that participants suggested very few implementation alternatives (from table 4.26: Group 1, mean=2.73, SD = 1.9; Group 2, mean 2.45, SD = 1.64). There were no differences between the groups, in the number of implementation alternatives suggested. It was originally hoped that participants of the ProtoTour group would have a clearer understanding of HCI design intent and would therefore be in a better position to propose various implementation alternatives. However, the general lack of alternatives proposed may have had another cause – 'satisficing' (Simon, 1981). Empirical evidence suggests that designers, when faced with an acceptable design solution, tend to stick with this after only moderate exploration of the solution space (Ball, Maskill and Ormerod, 1998). Revisiting the alternatives proposed by the participants reveals only two radical alternatives to the Checklist Window (one from each group). Most alternatives proposed were refinements on the design already shown in the prototype/ProtoTour. Clearly, this is consistent with the occurrence of satisficing. However, it could also be further evidence that programmers are unused to questioning the rationale for designs (as described in the discussion of research question 2 – section 4.5.3). It seems likely that both the visual prototype and ProtoTour interventions presented a solution that was acceptable to the programmers, so they did not explore the solution space further. Because of the possibility of becoming fixated with a design shown in a prototype without exploring the alternatives, the 'father of Visual Basic', Alan Cooper, has expressed his dismay that Visual Basic is so often used in this way, suggesting that 'paper prototyping' is a far more appropriate technique (Cooper, 1994).

It was further expected that the number of suggested alternatives relating to usability (Q3.4(ii)) and the number relating to improving the implementation (Q3.4(iii), would be found to be different between the experimental groups. However, this result was not found. Again, this lack of results is probably due to satisficing.

Experimental instrument: Participant opinion rating scales and open-ended questions

Data from these measures was considered unreliable and contained a large proportion of missing values.

These measures were intended to capture participants' rating of the usefulness of design rationale: in coming up with implementation alternatives (Q4.8); and in assessing those alternatives (Q4.9). As explained in the discussion of research question 2 (section 4.5.3), five participants from Group 1, claimed to have gained access to information about design rationale, when they had not. It is clear then that participants misunderstood the meaning of design rationale (reasons for this are covered in the discussion of research question 2). Therefore, data pertaining to design rationale is considered unreliable. Furthermore, there were six missing items from each experimental group on both of these measures which indicates that the majority of programmers did not understand what 'design rationale' referred to.

There were too many missing values in the data sets relating to these experimental measures to carry out meaningful statistical analysis.

Conclusion

These results provide no statistically significant evidence that the extra information in ProtoTour (including design rationale) enabled programmers to offer improved implementation alternatives than if they just had access to a visual prototype and an HCI Designer. In fact, participant performance against objective measures may indicate a serious concern about the use of prototypes or ProtoTour in a context where design ideas are sought.

Expert assessed measures of participant task performance proved ineffective. It is likely that this was for the same reasons outlined in section 4.5.2, i.e. the lack of realism of the tasks, especially the time constraints. However, the absence significant difference between the quality of implementation alternatives proposed by participants could be viewed as a positive result for ProtoTour. This is because the alternatives proposed by the ProtoTour participants were as good as those proposed by the visual prototype participants, even though they asked significantly fewer questions of the HCI designer about HCI design intent.

The objective measures employed to count the number of implementation alternatives proposed were also ineffective. This was primarily due to the very few alternatives that participants proposed. The possible cause of this, as well as the distinct lack of radical alternatives to the Checklist Window, leads to a serious concern about the utility of ProtoTour. The lack of radical alternatives could be explained by programmers being unused to questioning a design, other than to refine it (this is related to the discussion of programmers and design rationale in section 4.5.3). However, it seems likely that 'satisficing' is the primary contributor to this effect. If 'satisficing' does indeed cause programmers to focus on the solution proposed, rather than examining the rest of the solution space, there are implications for how ProtoTour should be deployed. If it is deployed too early in the software production process, when design ideas from programmers are welcomed, it may block them from having ideas about alternative designs. An approach, such as 'paper prototyping', which is thought to encourage the generation of ideas, may be a prerequisite activity to the production of a prototype or a ProtoTour representation.

Participants' rating of the usefulness of design rationale was considered unreliable, as there was evidence that participants misunderstood the term. Furthermore, these measures contained a large proportion of missing values – probably because programmers did not understand what the term 'design rationale' meant. Together, these factors precluded the analysis of the participant ratings.

As with research question 1, it is likely that the value of the extra information in ProtoTour would be better assessed by analysing programs built by programmers with this information, and comparing them with programs built by programmers using the more conventional approach. The artificial task and time constraints of the experiment did not provide a genuine means of assessing the research question. However, it is hard to see how the proposed more appropriate experiment could be viable, and how this could be devised to provide a useful comparison of ProtoTour and the conventional prototype approach. This is primarily because of the prohibitive cost of involving a large enough sample of commercial programmers in an experiment that would require at least five days of programming effort to implement something realistic that could be assessed. Furthermore, there would be difficult practicalities of running an experiment with more than 20 programmers working on the same thing for five days. Another practical difficulty would be the lack of commercial value of the tangible results of programmers' efforts over this time – 20 implementations of the same thing is unlikely to have much commercial value. Assessing new approaches in software development is a notoriously difficult problem (see section 4.2.2.2).

4.5.5 Discussion of Research Question 4

Research Question 4. - Does ProtoTour reduce the amount of time that HCI Designers need to spend explaining aspects of the visual prototype to programmers at their request?

This research question aimed to gain an indication of whether ProtoTour would reduce the amount of time that an HCI designer would spend explaining aspects of the design to programmers. Current practice showed there to be a lot of communication between the HCI designer and the programmer surrounding the visual prototype.

Two experimental instruments were designed to assess this research question: observation during the experiment; and participant opinion rating scales and open-ended questions. The experimental observation comprised counting and categorising the number of questions asked by participants during the completion of the design tasks. To support this instrument, participants were asked to rate the usefulness of the HCI designer during the experiment.

Experimental instrument: Observation during the experiment

One measure in this category detected significance at the (P<0.05) level but this is considered a weak result in this experiment. An experimental control measure was utilised to test whether one experimental group had a greater propensity to question than the other. The control measure found no differences between the groups, so this adds some support for the weak significant result found. Additional findings of this aspect of the study are derived from the categories in which only few questions were asked by participants.

The control measure (Obs. 3(v)) made use of the fact that participants of both experimental groups received almost identical instructions to ascertain whether one group had a greater propensity to ask questions than the other. This measure found no significant difference between groups on participants' questions about the task they had been set. Thus, the participants' propensity to question was considered the same between groups.

There was no significant difference between the number of questions asked concerning the usability of the visual prototype (mean = 0.45) versus that of ProtoTour (mean = 0.27). This result is slightly surprising as the visual prototype was a typical 'chauffeured prototype' (Preece et. al, 1994), which meant that it had been designed to provide an image of the proposed system in the hands of a trained operator. Without knowledge of how to 'run' this façade, it was expected that programmers would have difficulty making it 'work' appropriately. Programmers probably didn't report usability problems because they were used to wrestling with obscure and difficult software, and the façade was relatively robust. Thus, they asked no more questions about the usability of the intervention than the ProtoTour participants. ProtoTour participants were presented with an automated 'tour' of the prototype, with associated explanation and were expected to ask fewer questions about its usability. From the low mean counts, it is apparent that programmers from both groups asked very few questions about the usability of the intervention – perhaps programmers are used to dealing with software in this way.

Group 1 participants asked significantly ($p=0.039$) more questions about the intended functionality (HCI design intent) of the software than those of Group 2 (Obs 3(ii)). In this experiment, α levels < 0.05 are treated as marginal because of the large number of *t*-tests being carried out. Truly significant results were sought at $\alpha < 0.01$. However, as it is also estimated that around 16% of the variance has been accounted for, it does seem probable that the result does have some weight. Furthermore, the experimental control (Obs 3(v)) found that participants of both groups had a similar propensity to ask questions, which adds weight to any significant results discovered on similar measures.

Therefore, **there is some evidence that Group 2 (ProtoTour) participants asked less questions of the experimenter** (HCI designer), regarding the intended functionality (HCI design intent) of the software portrayed, than participants of Group 1 (prototype). There are perhaps two reasons for this. Firstly, it may be that ProtoTour does truly reduce the number of questions that programmers need to ask the HCI designer. ProtoTour provides an automated walkthrough and supporting explanation to enable programmers to build up a mental picture of the software portrayed. The visual prototype by contrast is hard to 'run' in the hands of someone not trained in its use and does not contain supporting information (this was provided by the HCI designer). Thus, there is some evidence that ProtoTour may provide 'better' information to programmers, reducing their need to ask questions. The second explanation is that Group 2 participants had more information to contend with, which did not leave them any time to ask extra questions. Participants of the other group may have explored the extent of the visual prototype, and still had time and spare capacity to ask further questions of the experimenter (the HCI designer).

Further analysis of data relating to Obs 3(ii) shows that the variance in the number of questions participants asked about intended functionality (HCI design intent) was significantly less in the ProtoTour group (Group 1: Mean 8.09; SD 6.99. Group 2: Mean 2.9; SD 3.39). This possibly provides some additional support for the finding that ProtoTour participants had fewer questions to ask in this category. Perhaps ProtoTour answered some fundamental questions very clearly, so no participants in that group had to vocalise them. Thus, the overall number of questions asked by ProtoTour participants was low. Whereas, the variation in the number of questions the prototype group participants asked was more in accordance with current practice (which Group 1 was intended to simulate). The simulation of current practice perhaps led programmers to behave more normally, some making assumptions and getting on with their job, and others asking lots of questions. Such variations in programmer performance have come to be expected in such studies due to individual differences (see section 1.10.2.1). Perhaps this is another indication that ProtoTour has potential for reducing the impact of individual performance differences among programmers.

Obs 3(iii) records the fact that no programmers asked about the design rationale for the software portrayed in the prototype. The discussion of this result associated with the discussion of research question 2 (section 4.5.3) relating to Q4.7, to which five participants claimed to have had access to design rationale information. These participants were from Group 1 where the intervention itself did not contain such information. Considered in the light of the results of Obs 3(iii), it had to be concluded that they were mistaken.

It is germane to ask why none of the 22 commercial programmers in the experiment asked a question about the rationale for the HCI design of the Checklist Window. This result seems to suggest that programmers take whatever information they are given and make a program out of it. Questioning the information, or the design rationale, appears to be something they have learned not to do. The extra information they did ask for during the experiment was often to confirm the design intent (how an aspect of the functionality was supposed to work), rather questioning why it worked that way. It is perhaps the programmers' **expectations** that has made them stop questioning 'why'.

Observation 3(iv) shows that from the 22 participants in the study, only one (from Group 1) asked a question about users and/or their tasks. However, ProtoTour did contain some relevant information about users and tasks so some of the programmers from Group 2 may have accessed such information (although from observation during the experiment, this did not appear to be the case). In general, it would appear that programmers in the study did not find it necessary to ask questions about users and tasks. There are a number of possible reasons for this. As research suggests (see section 1.10.2), this may be because programmers are not very good at thinking about users. Some certainly appear to have a naïve belief that a user interface design can be inherently good and ignore the fact that a design has to be assessed in the context of who will be using it. This may stem from the fact that the design of the internal structure of a computer program can be regarded as inherently good. A common mistake of HCI designers is to design as if they are the user, instead of thinking about the users (Heckel, 1991). Perhaps this experimental result provides empirical evidence that programmers do design for themselves, and fail to consider how users may be different from them. Another plausible explanation for the lack of questions about users and tasks could be that in the context of this HCI experiment, participants assumed the user interface design had been taken care of. However, this seems unlikely when the results from Q3.4a(ii)-(iii) are considered. These results show that programmers in each group proposed more implementation alternatives relating to usability, than they did alternatives to ease implementation. Table 4.45 (derived from tables 4.27 and 4.28) shows that participants generally suggested more usability alternatives than alternatives relating to implementation.

Table 4.45 Comparison of the number of alternatives relating to usability and those relating to implementation

	Q3.4a(i) Alternatives relating to Usability			Q3.4a(ii) Alternatives relating to implementation		
	N (participants)	**Mean**	**SD**	**N** (participants)	**Mean**	**SD**
Group 1 – visual prototype	11	2.4545	2.018	11	.3636	.505
Group 2 – ProtoTour	11	1.3636	2.063	11	1.0909	1.136

A precise comparison is not relevant, but these measures clearly show that programmers do make suggestions relating to usability, and yet only one participant from the 22, asked a question about users or their tasks. This is particularly interesting in the case of the Group 1 (prototype), as they received no information about users from the prototype itself (unlike the ProtoTour participants). However, the mean number of usability alternatives proposed by participants in this group was 2.46, compared with only 0.36 relating to the actual implementation. This would seem to support the hypothesis that some programmers believe that a user interface can be inherently good, rather than understanding that a design must be evaluated in its context of use.

This finding may also illustrate a certain role ambiguity that programmers suffer when an HCI designer owns the user interface design. It was expected that programmers would focus their suggestions on how the implementation could be simplified by changing the design, as this would relate directly to their role. However, perhaps for historic reasons, or maybe out of a desire to express their creativity (see section 1.10.2), the participants choose to focus on suggesting alternatives to improve usability. This is an aspect of the software development that prototype-centred interventions clearly suggest they do not need to be involved in, and yet many focused effort here instead of on their part of the job, the implementation. Perhaps programmers like to rise to the challenge of improving user interface design ideas. As the topic of a long-term study of a software project, it would be revealing to analyse just how many usability versus implementation 'alternatives' programmers come up with in practice. It is likely that when designing on paper, programmers are unable to foresee aspects of the user interface design that will be difficult to implement. However, in practice, as they begin to write the code they will hit these problems and are likely to start suggesting alternatives at that point. The slightly unrealistic nature of the experiment perhaps also contributed to the lack of suggested implementation alternatives.

Experimental instrument: Participant opinion rating scales and open-ended questions

Participants were asked to rate the usefulness of the HCI designer during the experiment (Q4.6(iii)). It was expected that this would provide an indicator of the degree to which participants felt that they did not need the presence of an HCI designer to support the intervention. There was no significant difference between the ratings of the two experimental groups. This measure may have been flawed because the post-test questionnaires were completed in the presence of the HCI designer. Thus, all participants may have been reluctant to score this contribution negatively.

Conclusion

There is evidence to suggest that visual prototype participants asked more questions regarding the intended functionality (HCI design intent) of the intervention, than did ProtoTour participants. This result was significant at p=0.039, which can only be considered statistically marginal significance because the experiment utilised a large number of *t*-tests (thus increasing the probability of making a type I error). However, a control measure was utilised in this aspect of the experiment. The control, which measured participants propensity to ask questions, found no significant differences between the groups. This adds weight to the finding that visual prototype participants asked more questions about intended functionality. Yet more weight is added to this discrimination by the fact that there was less variance in the number of questions asked by participants from the ProtoTour category. Thus, it could be the case that ProtoTour answered some fundamental questions for all programmers in that group. However, programmers within the visual prototype group are thought to have behaved more naturally, some asking lots of questions, and others making lots of assumptions. Thus, ProtoTour again seems to have potential for reducing the affects of individual differences in programmers' performance. These findings contribute to an indication that ProtoTour could indeed reduce the amount of questions that an HCI designer is asked regarding intended functionality (HCI design intent).

No participants asked a question about design rationale and only one participant asked a question relating to users and their tasks. Participants in the ProtoTour group did have access to such information, but there is some doubt about whether it was utilised. Perhaps these findings illustrate the current expectations and role of programmers in commercial practice. Programmers may not question 'why' a user interface design is as it is because it is their job to implement it. These findings may also illustrate that programmers have a naïve attitude towards user interfaces. Although only one question was asked about users and their tasks, most programmers proposed implementation alternatives to improve usability. This is evidence that programmers believe that a user interface can be inherently good without considering its intended use context with its intended users.

Furthermore, the finding that most participants suggested more implementation alternatives to improve usability than they did to improve the implementation itself perhaps reveals a certain role ambiguity. This may be because most programmers in the study have never worked with a user interface designer and so are still responsible for this aspect of the system. However, the prototype-centred interventions presented programmers with a completed user interface design so it would not be unreasonable to expect them to have focused on their aspect of the job – the implementation.

It is thought that programmers would actually come up with more implementation alternatives than usability alternatives in practice. On paper, it is hard for them to foresee implementation problems. In practice, they have no opportunity but to suggest alternatives when problems arise. However, a future study would be needed to further assess this finding and verify the hypothesis.

The participant rating of the usefulness of the HCI designer is considered a flawed measure. It is felt that all participants would have been reluctant to be negative about the HCI designer's contribution during the experiment. Hence, there was no significant difference between the participants rating of the HCI designer between the groups.

4.5.6 Discussion of Qualitative Results

Question Q4.11 of the post-test questionnaire asked participants for comments and criticisms about the intervention they had used. This discussion is based on the categorisation (post hoc) of these qualitative results (see results section 4.4.6).

The following table 4.46 summarises the main categories of comments made. This qualitative information should not be over-analysed, but there are a few features worthy of particular comment.

Table 4.46 Collection of Qualitative Results - Participants' Comments

Category of Comment	**Number from Group 1** (n=comments per category)	**Number from Group 2** (n=comments per category)
need more information	(x4) e.g. "a visual prototype is not sufficient in itself to fully describe a design."	(x2) e.g. "Some information on amounts of data, sizes of text, etc. were not included."
need an HCI designer for further explanation	(x1) e.g. "During the exercise this problem [the prototype gave no information about underlying functionality] was overcome by the HCI Designer providing extra information verbally."	(x3) e.g. "I wouldn't solely want the ProtoTour as an illustration, but combined with liaison with the HCI designer looks pretty good!"
problems of increasing expectations of users and clients by using a prototype approach	(x4) e.g. "Sometimes visual prototyping can mislead the customer into thinking they already have a close to working system when most of the functionality does not in actual fact exist."	(x0)
need for the programmer and HCI designer to work together	(x3) e.g. "HCI Designer might not understand what is required to actually implement the design."	(x1) e.g. "A general criticism would be that designers and programmers should always work together to ensure "realistic" demos."
concern about cost and effort required to construct ProtoTour	(x0)	(x3) e.g. "I have doubts about the effort it must take to produce a ProtoTour..."
usability of prototype/ ProtoTour	(x1 criticism) e.g. "I wasn't completely clear how the Checklist was supposed to work..."	(x3 criticisms; x1 positive comment) e.g. "In the timed exercise I felt there was too much information available and I felt a bit swamped..."
prototypes are for getting requirements right	(x3) e.g. "Prototype helps to capture user requirements accurately..."	(x1) e.g. "I think the main use of these prototypes is to investigate how the user sees the problem. "
other positive comments about the prototype or ProtoTour	(x5) e.g. "The visual prototype was good for showing the appearance and behaviour of the interface..."	(x5) e.g. "Yeah, it's good... In the timed exercise I felt there was too much information available and I felt a bit swamped but in a real situation this information would prove invaluable. It serves to support the HCI Designer's view and gives the implementor confidence that the HCI Designer has thought through the issues with rigour. "

Four programmers using the visual prototype made comments expressing concern about the issue of increasing user and client expectations that can arise when using visual prototypes. Nobody in the ProtoTour experimental group made any comments about this issue. Perhaps this is an indication that ProtoTour was perceived as unlikely to convey unrealistic expectations to the non-technical audience. Alternatively, programmers may have perceived ProtoTour as something that is not intended for a non-technical audience; therefore, concern about expectations was not at the forefront of their minds.

Three programmers using ProtoTour expressed concern about the effort that would be required to put together a ProtoTour representation. Nobody in the visual prototype group expressed any similar concern about the construction of the visual prototype. This criticism of ProtoTour is valid and is discussed in section 2.10.4.10.

The summary of general positive comments shows an equal number (five) were made for the visual prototype and for ProtoTour. This provides further support for the results of question Q4.12 that found 20 programmers across the experiment (91%) to be in favour of prototype-centred explanation (10 specifically rated the visual prototype and 10 rated ProtoTour). Programmers certainly appear to be positive about the use a prototype-centred intervention.

4.6 Summary of Findings

This experiment has provided evidence that programmers are able to gain a better understanding of HCI design intent using ProtoTour than through use of a visual prototype (research question 1). The strongest evidence of this has come from three post-test comprehension-style questions about the intended functionality (HCI design intent) of the Checklist Window. Participants from the ProtoTour group answered these questions with significantly greater accuracy (P<0.005) than those from the visual prototype group. Expert assessments of participant task performance found that there was significantly less variance in the task performance of ProtoTour group participants on three measures (although there was no significant difference in mean performance between the groups). Across these measures, the lower performers in the ProtoTour group scored higher than the lower performers in the visual prototype group. Thus, there is some indication that ProtoTour could have a role in reducing the effects of individual performance differences between programmers. Although this is a weak result, this is worthy of further research, as mixed ability teams are believed to be commonplace in commercial software teams.

There was no evidence to suggest that programmers would find ProtoTour more accessible than a visual prototype (research question 2). However, prototype-centred explanation was rated favourably by 10 of the 11 participants in each group and five participants from each group made favourable comments about the intervention. This provides an indication that programmers are generally in favour of prototype-centred approach. The appreciation of the approach is likely to led to an acceptance of it in practice, which is a favourable indication for its likely accessibility.

There was no evidence to suggest that the extra information in ProtoTour would enable programmers to offer improved implementation alternatives than if they just had a visual prototype (research question 3). However, a potentially serious problem with the use of prototype-centred explanation has been indicated by the small number of alternatives suggested by participants, combined with the finding that only two radical alternatives to the Checklist Window were suggested. It appears that prototype-centred explanation could potentially led to ‘satisficing’. This may cause programmers to focus on making improvements to the presented design, rather than exploring radical alternatives. Thus, if deployed too early, prototype-centred explanation could repress the programmers’ contribution to the design process.

There is evidence to suggest that ProtoTour would reduce the amount of time the HCI designer would have to spend explaining aspects of the visual prototype to programmers (research question 4). ProtoTour group participants asked significantly fewer (P=0.039) questions about intended functionality (HCI design intent) than those of the visual prototype group. This level of significance is considered marginal due the number of t-tests carried out. However, a control measure found there to be no significant difference in each group’s propensity to ask questions, thus providing support for the finding.

Considered together, some findings from the experiment begin to demonstrate the potential of ProtoTour. Of particular interest is the finding that ProtoTour participants asked the HCI designer fewer questions about HCI design intent and yet they answered post-test comprehension questions with significantly greater (P<0.005) accuracy than visual prototype participants. Furthermore, the implementation alternatives proposed by programmers using ProtoTour, were as good as those suggested by programmers from the visual prototype group that asked significantly more questions of the HCI designer. This indicates that ProtoTour could reduce the amount of explanation an HCI designer would have to provide to support the visual prototype and that programmers would get a better understanding of HCI design intent with this approach.

A number of findings from this experiment relate to programmers. A number of programmers suggested that their experience of programming exceeded their user interface programming experience by a number of years (on the pre-test). It is thought that this is because some programmers do not understand the distinction between a user interface and a GUI. This misunderstanding has implications for how they perceive the role of the HCI designer. Although this appears to be a speculative conclusion, there was further evidence to support this. Only one programmer in the experiment asked a question about users and their tasks and yet most programmers specified more than one design alternative to improve usability. This suggests that programmers perceive usability to be an inherent attribute of software, rather than a context dependent consideration. In fact, programmers suggested more alternatives relating to usability than implementation. This was unexpected because the programmer's role is more to do with implementation. These findings seem to illustrate that the role of the HCI designer is unclear to programmers. Their performance indicates that they perceived their role to have some degree of responsibility for the user interface. Findings relating to design rationale are also of concern. No programmers asked questions about the rationale for the design and yet five of them thought they had received such information when they had not. Although this lack of questioning is likely to have been contributed to by the artificial nature of the experiment, the finding is too strong to be completely explained by this. It is thought that programmers have learned to take whatever 'input' is given and make a program out of it, without questioning. Perhaps programmers' expectations about the user interface have formed in such a way that they do not perceive it possible to question the reason for the design. This could have implications on the worth of providing design rationale information.

Individual differences among programmers were also evident. There was considerable variability in the design representations selected by the programmers to depict their preliminary designs (seven different types of representation used in all). Time estimates for implementation of the Checklist Window were highly variable, ranging from 1 to 25 days, with a mean of 7.5 and a standard deviation of 7.0. On several measures, ProtoTour participants were seen to have significantly less variance in their assessed task performance. In all cases, the use of ProtoTour seemed to have lifted the performance of the lower performing programmers, although the result was no significant difference in mean performance between the groups. If ProtoTour could indeed improve the performance of the lower performing programmers in a mixed ability team, it could be worthwhile in commercial software practice where misunderstandings can prove costly.

Although this experiment design is still considered the most appropriate to deriving a rigorous comparison between the ProtoTour and visual prototype interventions, some aspects were ineffectively assessed through this approach. Throughout the experiment, expert assessed measures were ineffective. This was probably because the artificial nature of the experiment only allowed the expert to assess the very early stages of the programmers' implementation tasks. At this stage, before any code has been written, it is difficult for programmers to foresee the real implementation difficulties that will occur. An experiment that facilitated assessment of programs produced, based on the intervention utilised by the programmers, would be a far better test of whether they had understood HCI design intent correctly and whether they had utilised appropriate alternatives. A participant-observation study of the interventions being used in practice would be an appropriate way to further assess their practical worth, as well as their accessibility. However, using this approach for comparison of the interventions would be difficult and qualitative. Assessing new approaches in software development is recognised as a notoriously difficult problem (Bellotti, 1990; Carey et. al, 1991; Harker, 1991).

4.7 Overall Conclusions

Findings relating to the research questions of this experiment are concluded below, with a final analysis bringing together conclusions across the research questions.

Research Question 1: Does ProtoTour convey a better understanding of HCI design intent (i.e. less ambiguous) than a visual prototype?

- Results suggest that programmers are able to gain a better understanding of HCI design intent using ProtoTour than through use of a visual prototype.

- Task performance of programmers using ProtoTour was found to be less variable than that of programmers using a visual prototype, with fewer low performance scores.

Research Question 2: Does ProtoTour improve the accessibility of the visual prototype?

- There were no experimental results to suggest that programmers would find ProtoTour any more accessible than a visual prototype.

- Findings highlight the fact that most programmers are in favour of a prototype-centred approach to conveying HCI design intent (i.e. an approach such as a visual prototype or ProtoTour).

Research Question 3: Does the provision of extra information in ProtoTour (including design rationale) enable programmers to offer improved implementation alternatives than if they just had access to a visual prototype and an HCI designer?

- There were no experimental results to suggest that programmers using ProtoTour would offer improved implementation alternatives than if they just used a visual prototype.

- Findings suggest that both the visual prototype and ProtoTour could lead to ‘satisficing’, which could repress the contribution of programmers to the design process.

Research Question 4: Does ProtoTour reduce the amount of time that HCI Designers need to spend explaining aspects of the visual prototype to programmers at their request?

- Results suggest that the use of ProtoTour could reduce the amount of time that the HCI designer would have to spend explaining aspects of the visual prototype to programmers.

Final Analysis

The concept of prototype-centred explanation embodied by ProtoTour has been found to be of great potential benefit to software development. Using ProtoTour, programmers appear to have less need to ask questions of the HCI designer, and yet develop a better understanding of HCI design intent, than if they use a visual prototype supported by explanation by the HCI designer. Furthermore, implementation alternatives proposed by programmers using ProtoTour, appear to be as good as alternatives that would be proposed if they used a visual prototype and asked an HCI designer for further information. Thus, ProtoTour could provide a means of adequately supporting software development without the need for a human HCI expert to be available to all programmers, at all times.

One important drawback of the use of ProtoTour or visual prototypes is that satisficing can occur, blocking a programmer's ability to contribute design alternatives. This could have a serious effect on a software development by repressing the input of the software technology experts in the team. Thus, the deployment of ProtoTour needs to acknowledge this problem.

Chapter 5 **Discussion**

5.1 Current Software Production Conditions

There is clearly something wrong with the way in which much software is produced because there are reports in literature of a large proportion of software projects failing during production and acceptance (e.g. Grudin, 1993).

The Waterfall lifecycle is still dominant in the software industry, although many researchers and practitioners believe it to be an inappropriate method of facilitating software production (e.g. Boehm, 1988). It is argued in chapter 1, that its continued dominance is because it prescribes early specification and early estimation. Although it is recognised (by most researchers) that these will usually be inaccurate at best, this lifecycle at least provides some kind of answers to fundamental commercial questions like, 'how long will the software take to build?'. Other lifecycles recognise that such early specification and estimation is futile and refuse to prescribe these activities. Therefore, the Waterfall is seen as providing answers, whereas more appropriate iterative lifecycles, like RAD, do not.

Software production is regarded as strongly influenced by the nature of software development, which is chaotic (Craig, 1991), and susceptible to change (Miller-Jacobs, 1991). Similarly, the people involved in software production form another dimension of the nature of software development. For example, extreme individual differences in performance are widely reported amongst commercial computer programmers (e.g. Curtis *et al.*, 1987). One consequence of this, not reported in the literature, is that commercial software is often produced by mixed ability teams.

Commercial software production cycles and division of labour conspire to prevent programmers learning about the product of their labours. For example, programmers will often move on to another project before a product is shipped to users. Therefore, programmers usually never see people using their creation. Consequently, they never learn the inadequacies of the user interfaces they have produced.

HCI has not yet gained a significant foothold in commercial software development. It is apparent that most user interface design is still done by programmers on an *ad hoc* basis (Browne, 1994). Users are rarely involved in the design process in any way (Grudin, 1993). The software industry appears not to understand the contribution of HCI and human factors. Researchers agree that HCI theory is rarely applied in practice (Buckingham-Shum and Hammond, 1994). Where the need for HCI is recognised, it is often bought in on a consultancy basis, rather than as a fundamental activity of the software development team. Multi-disciplinary development teams are therefore not yet regarded as commonplace.

GUIs have caused more software development emphasis to be placed on the design and implementation of the user interface. However, this is largely because they are complex and require a greater proportion of development effort to build. The increase in resources required to design and implement GUIs (Myers and Rosson, 1992), has not led to an equivalent level of improvement in the quality of user interfaces.

Positive future directions in software production apparent from literature include:

- Use of Prototypes to:
  - bring about changes in software design early in the development process (even if they are not shown to users - Mantei and Teorey, 1988);
  - capture user requirements by providing a common representational currency between the developers and the users (Harker, 1991);
  - enable HCI specialists to take on a more dominant role in user interface design (Rudd and Isensee, 1994);
  - facilitate communication within a software team (Miller-Jacobs, 1991).

- The need to maintain conceptual integrity of design within the software team is recognised by several eminent researchers (e.g., Curtis *et al.*, 1987; Heckel, 1991; Norman, 1986), but little actual research has been conducted in this area (Brooks, 1995). However, the need to focus on comprehension within the team, in order to maintain a common shared understanding of the software being produced and conceptual integrity of design, has been identified.

- The increasing recognition of the fact that HCI theory is hard to apply in practice (Lansdale and Ormerod, 1994) and the suggestion that it could be considered a craft skill (Dowell and Long, 1989).

Current software production conditions and the positive directions identified guided the focus of this research.

5.2 Introducing an HCI Designer into a Commercial Software Team

Chapter 2 describes a detailed qualitative investigation of the effects of introducing an HCI designer as a full team member into commercial software projects. The HCI designer's usual role is often described as being a consultant to the team (Carey *et al.*, 1991; Coggman and Cohen, 1995; Lansdale and Ormerod, 1994; Stewart, 1991). Therefore, a primary motivation for this study was that it was believed that HCI designers rarely operated in teams in this way. However, there is recognition in the literature that it is inevitable that HCI designers will begin to take on roles like this in the future (Heckel, 1991; Kim, 1990; Norman – reported in Rheingold, 1990). The remainder of this section describes the practical implications of introducing an HCI designer into a commercial software development team.

The HCI designer beginning work in the team for the first time would be well advised to focus considerable attention on how the software team shares an understanding of the details of the software being built. In particular, how HCI design intent can be conveyed in such a way that all team members have a consistent understanding of it. Or in other words, how to maintain conceptual integrity of design throughout the team. On a practical level, an HCI designer new to a team would do well to carefully observe the nature of the software team and the commercial software development setting. The findings from this study suggest that there are a number of features of the nature of the software team and software development that are almost unavoidable 'facts' of the commercial setting. These are described in section 5.2.1 below. Features of the setting that aim to facilitate comprehension within the team (and with other stakeholders) are then described. Finally, this section highlights the interaction difficulties between the HCI designer and programmers in a software team.

5.2.1 The Nature of Commercial Software Teams and the Development Setting

Software teams appear to be dominated by programmers or people who have progressed from this role to similar kind of role, e.g. designer/programmer. Severe individual differences in the performance of commercial programmers is a well acknowledged finding (Curtis, 1981). Such findings usually relate specifically to the discrete work tasks of programmers, for example debugging a program or writing some code, measured in a quantitative experiment. The qualitative investigation of this study has found some evidence to support these findings, observing that the software team studied exhibited a mixture of programming abilities. Aspects of the programmer performance in the experiment described in chapter 4 provide quantitative support for individual differences among programmers. When asked to estimate how long it would take to implement a particular window they had been working on, programmers' estimates ranged from one to 25 days with a mean of 7.5 days and a standard deviation of 7.0 days. Programmers also demonstrated considerable diversity in the design representations they adopted, with seven different approaches in evidence from a sample of 22 commercial programmers. This diversity in design representations is similar to Curtis *et al.*'s finding (1987), which reported tremendous variability in how programmers carried out tasks.

An additional finding from the qualitative investigation is that programmers seem to have a range of other characteristics which differ strongly. For example, their ability to understand (conceptualise), visualise and explain software seemed diverse. This relates to suggestions from Curtis *et al.*(1987) and Brooks (1986) that some rare team members have exceptional conceptualisation ability, which Curtis *et al.* labelled 'super conceptualisers'. Thus, from a practical perspective, it would be beneficial to an HCI designer entering the field to ascertain the particular strengths and weaknesses of the team members with programming-related roles. For example, a programmer particularly skilled with syntax may be an appropriate person to help determine whether an HCI design concept could be implemented with the technology available.

Another aspect of the fundamental nature of the software team, is that team members from different disciplines experience problems understanding each other. The HCI designer entering a commercial software team is likely to notice this with programmers in particular. There are reports in literature of such difficulties in multi-disciplinary teams (Karat, 1996; Kim, 1990). However, multi-disciplinary teams are not yet believed to be widespread. It appears quite likely that the HCI designer could be the first team member to enter a team, with a disciplinary background other than programming. Therefore, these inter-disciplinary misunderstandings seem to be prevalent in the HCI designer's interaction with programmers. For example, from the qualitative investigation, it was apparent that some programmers failed to appreciate the fundamental importance of certain HCI design rationale that related back to the users' tasks; they were pre-occupied with the technical difficulties that such an acknowledgement would cause. Thus the HCI designer's focus on the needs of the users and the programmers' focus on the 'needs' of the technology could cause misunderstandings and conflict.

From the results of the experiment described in chapter 4, it is apparent that multi-disciplinary issues and role ambiguity can arise from the programmer's perceptions of user interfaces, usability and the role of the HCI designer. Some programmers taking part in the experiment reported many years programming experience and only a few years experience programming user interfaces. It seems that these programmers believed that there was no such thing as a user interface before the advent of the GUI. This supposition is also supported from my recall of numerous repetitive discussions about this with programmers during the fieldwork. This misunderstanding of what a user interface is can clearly led to misconceptions about the role of the HCI designer in the team. Further results from the experiment support the concern that programmers have a poor perception of user interfaces and usability. For example, none of the 22 programmers in the experiment questioned the rationale for the HCI design they were presented with, and only one person asked a question about the users and their tasks. Nonetheless, most programmers suggested more design alternatives aimed at improving usability than they did for improving the implementation. Thus, it seemed that programmers taking part in the experiment perceived usability to be an inherent feature of software, rather than a context dependent feature. The programmers' focus on design alternatives to improve usability over design alternatives to make implementation more efficient suggests that role ambiguity is an issue. The introduction of an HCI designer into a software team will lead to a necessary change in the role of programmers. It is therefore inevitable that part of the HCI designer's initial role in the team will be educating team members as to the boundaries of their role, and by implication, the change in their roles. Billingsley (1988) found that programmers are often proud of their interface designs and resent threats to this means of expressing their creativity. The introduction of an HCI designer taking overall responsibility for the user interface is clearly an extreme case of removing a creative outlet from programmers.

The HCI designer entering the setting should expect the composition of the development team to change over the duration of the project (Grudin, 1996; Harker, 1991). It appears that it is a common occurrence for programmers to join the team following the completion of the specification of the software. These new team members create a comprehension burden on the team, and the HCI designer in particular, who is likely to be responsible for conveying HCI design intent to them. Although intended as a means of informing various stakeholders about what the software being built will do, the specification document itself, appears to be compromised in this role because of the diversity of stakeholders it must serve. Much of the literature suggests that specification documents are an insufficient communication medium for design concepts (Grudin, 1993; Muller, 1993; Curtis *et al.* 1987). Therefore, the HCI designer can expect to devote time to explaining the proposed software to other team members, users and management. One tool that could be used to facilitate these explanations is visual prototypes. Initially created to elicit user requirements, they ultimately become a concretisation of the HCI designer's conceptual model of the proposed software. It is believed that visual prototypes could be the future advance, which Curtis (1988) suggested was necessary to bridge the gap between the statement of requirements and software design.

A further inevitable feature of the commercial software development setting is that changes to requirements will occur throughout the duration of the project. This is well recognised in literature (Beladay and Lehman, 1979; Brooks, 1986; Miller-Jacobs, 1991) and was in evidence throughout this study. One problem with changing requirements is that they must be conveyed to the development team who are often already swamped with paper. Therefore, poor communication of changes in requirements is likely to cause team members to have inconsistent understandings of the proposed software. Although changing requirements are regarded as inevitable in existing literature, the consequences that this has on the shared understanding of team members is not reported.

5.2.2 Aspects of Software Development that Aim to Facilitate Comprehension

The HCI designer entering a software team is likely to find a number of aspects of the work in the setting that relate to facilitating comprehension of the software under construction.

Software engineering representations are utilised by programmers and software designers to represent the design of the internal structure of the software. Such representations are not readily accessible or understandable to those without experience and training in their use. These representations have been around a long time in software engineering and are rarely regarded as a powerful means of facilitating communication among programmers. In fact, the underlying thesis of Brooks' (1975) 'The Mythical Man-Month' is that people and months are not interchangeable (on the project plan), because of problems associated with communication. However, although such representations are of limited use in facilitating understanding, even amongst programmers, the HCI designer does have something to gain by becoming familiar with them. The representations provide a view on the programmers' understanding of the form of the software under construction, so it was found to be useful for the HCI designer to be able to comprehend these representations. By this means, the HCI designer could ensure that team members had understood HCI design intent and that their design of the internal structure of the software was appropriate.

The HCI designer entering a new software team would be wise to recognise the number of small decisions that programmers make on a day-to-day basis during the implementation phase. Existing literature does not warn of these decisions or possible causes of poor decision-making. From the qualitative investigation it was apparent that many programmers did not have a good general understanding of the software project they were working on (e.g. they did not understand the objectives for the project). In fact, several of the programmers studied appeared to see this as something that was outside the scope of their role. Nonetheless, these programmers all made day-to-day decisions about the shape of the software. In practice, the HCI designer would be well advised to make it their business to ensure that the programmers understood some basic background about the project in order to improve their decision-making. Furthermore, this issue indicates the need for the HCI designer to maintain a 'watching brief' on the software as the implementation progresses.

Software specifications are emphasised by the waterfall lifecycle (Grudin, 1991), which is believed to be the most prevalent software development process. Although software specifications appear to be an inevitable feature of the commercial software development setting, and are designed to facilitate comprehension, they are compromised by their diverse uses and readership. The HCI designer entering the commercial software development arena should carefully consider the appropriateness of expending a large amount of effort contributing to the specification document. Instead, they could consider other more appropriate means of conveying HCI design intent to the user and the development team. For example, some researchers suggest that visual prototypes could be used to support and explain what is written in specifications

(Gomaa, 1983; Heckel, 1991; Wilson and Rosenberg, 1988). This study verified this suggestion.

Visual prototypes were found to be an effective means of facilitating team members' visualisation of the software under construction, before a line of production code had been written. Such prototypes were constructed by the HCI designer in the projects studied, as a means of eliciting user requirements for the proposed software – this mode of use of prototypes is often described in the literature (Brooks, 1986; Luff, Heath and Greatbatch, 1994; Overmyer, 1991). However, it soon became apparent that such prototypes had great potential within the development team to help team members to share a consistent conceptualisation of the software under construction. This use of prototypes has not been widely reported in existing literature but Miller-Jacobs (1991) and Wagner (1990) suggested that they might serve this purpose very well.

Visual prototypes were found to have great utility for conveying HCI design intent within the team. This use of an existing prototype should be seriously considered by any HCI designer. However, this study found several flaws with using prototypes in this way, which are not acknowledged in the literature:

- they can be ambiguous and open to interpretation, because it is not clear from the prototype which elements reflect a well-designed concept and which elements have been mocked-up to complete the picture;

- they fail to explain the HCI design rationale for the proposed software shown in the prototype, providing programmers with no basis from which to negotiate or offer design alternatives (which may have been easier to implement or provided a better solution);

- because the prototypes used in the qualitative investigation were chauffeured-prototypes (Preece *et al.*, 1994), they were primarily designed to be operated by the prototype builder, utilising discreet key presses and hidden locations on screen, thus they were somewhat inaccessible to programmers.

Ultimately, the qualitative investigation focused on these aspects of software development that aim to facilitate comprehension, as these were believed to be the best targets for improving comprehension within the team. This focus led to the finding that all of these targets had a central position in the interaction between the HCI designer and the programmers, and specifically, how they shared an understanding of the design. Thus, the final focus of this funnel stage was the interaction between these roles and the potential comprehension difficulties that existed between them.

5.2.3 Difficulties in the Interaction Between the HCI Designer and Programmers

Re-focusing the investigation onto the HCI designer and the programmers identified specific difficulties in the interaction between them. These difficulties and their implications for current software practice are described below.

1. Programmers misunderstanding or misinterpreting HCI design intent

Misunderstandings and misinterpretation of HCI design intent by programmers can cause a number of serious problems in software production. If possible, an HCI designer in a commercial team should maintain a 'watching brief' over the work of programmers, in order to ensure that misunderstandings do not occur.

This study has found a number of potential causes of misunderstanding and misinterpretation of HCI design intent:

- ambiguity and inaccessibility of prototypes;
- inadequacy of specifications and other representations;
- lack of explanation of design rationale;
- limitations of the abilities of individuals to conceptualise.

It is unsurprising that specifications give rise to misinterpretations and misunderstandings because their inadequacy as representations is well acknowledged in the literature (Curtis *et al.*, 1987; Grudin, 1993; Miller-Jacobs, 1991; Muller, 1993; Wilson and Rosenberg, 1988).

Although Conklin and Burgess-Yakemovic (1996) and Grudin (1996) also found design rationale ineffectively conveyed in software teams, they did not report specific effects of this.

Observed effects of misunderstandings and misinterpretations were:

- misdirected implementation requiring major rework;
- inappropriate day-to-day decisions made by programmers;
- programmers failing to appreciate the design importance of a particular feature, risking its omission from the implementation;
- delayed discovery of problems with a design, creating the need for re-design at a later stage;
- lengthy team meetings to clarify concepts.

2. Failure to maintain conceptual integrity

Maintaining conceptual integrity of design has been a problem in commercial software development for a long time, because of the complexities of software construction. These comprehension difficulties were apparent in teams comprised solely of programmers (e.g. Brooks, 1975), consequently it is unsurprising to discover this problem in a multi-disciplinary team.

In order to develop a high quality user interface to a product, an HCI designer should pay careful attention to maintaining conceptual integrity of design within the team. In practice, this means ensuring that all programmers share a consistent understanding of the product they are building.

Causes of a breakdown in conceptual integrity within the team include those that cause misunderstandings and misinterpretation, with several additions:

- uneven distribution of general project understanding and domain knowledge;
- poor communication within the team;
- changing composition of the team;
- inadequately conveyed changes to requirements.

Research into communication within the software team has been called for by a number of researchers (Curtis *et al.*, 1986; Erickson, 1995; Krasner, 1986; Mountfield, 1990), illustrating the lack of existing research in this area .

The effects of a breakdown in conceptual integrity are also the same as those reported for misunderstanding and misinterpretation, with one addition:

- software is pulled in different directions by those producing it, affecting its quality and usability.

3. Heavy Volume of Communication between HCI Designer and the Programmers

The HCI designer joining a software team should be prepared for a heavy volume of communication with programmers, particularly during the implementation stage. Often, this communication will be repetitive, clarifying the same details of HCI design intent to various programmers. Because programmers tend to have low needs for social interaction in the workplace (Couger and Zawacki, 1980), the HCI designer should not discourage them from asking for clarification on aspects of HCI design intent. In fact, the HCI designer should encourage programmers to ask whatever questions they need to because their only alternative is to make assumptions.

Causes of the heavy volume of communication between HCI designer and programmer include:

- inadequate project documentation;
- inaccessibility and ambiguity of prototypes;
- lack of recorded design rationale.

Literature relating to the inadequacy of specification documents and ineffective distribution of design rationale information within the team (mentioned under 1. in this section), is also relevant to this issue.

The consequences of this problem are:

- programmers having little alternative but to ask many questions of the HCI designer;
- disruption to the HCI designers work.

4. Ineffective Negotiation Between HCI Designer and Programmer

In current practice, it is usual for programmers to be uninformed of HCI design rationale. Furthermore, findings from the experiment described in chapter 4 suggest that programmers do not expect to be able to question the rationale for a given user interface design. None of the 22 programmers in the experiment took the opportunity to ask a question about design rationale during the experiment.

If programmers are not presented with HCI design rationale, it is impossible for them to tell which aspects of a design are well thought out and which aspects are not. However, if programmers are presented with design rationale, they are able to contribute alternative design ideas that meet the rationale, rather than blindly programming the given designs. It is this input from the technology experts that must be encouraged by the HCI designer, programmers and the project manager. A comment from a programmer involved in the experiment described in chapter 4, illustrates this point:

"Major criticism is that prototypes can guide implementation too much – coder [programmer] can lose sight of some implementation issues and may be forced to produce something that could have been implemented in a better way." (table 4.43, section 4.4.6)

In the current situation, because programmers are not informed of design rationale, they are not in a position to negotiate with the HCI designer over the implementation of an aspect of the design. This can cause several problems:

- programmers may strive to produce an implementation that is a faithful representation of the HCI design shown in the prototype, for an aspect of the design which could equally well have been implemented in the spirit of HCI design intent with a different easier-to-implement design;
- programmers may fail to understand that an apparently trivial aspect of the design shown in the prototype has a fundamental importance to HCI design intent;
- programmers may make wildly inappropriate design suggestions, because they are not aware of the design rationale;
- programmers may make inappropriate assumptions about how the design should be changed in order to better suit the implementation tools/technology.

Literature describing ineffective negotiation between HCI designers and programmers is not believed to exist. However, Moran and Carroll (1996) suggested that artifacts produced by the design process do not inherently indicate the reasoning underlying their design. So, it is unsurprising to discover that a flaw of the prototype-centred approach is that prototypes do not convey design rationale information.

5. Duplication of Effort

In commercial practice, programmers may duplicate the efforts of the HCI designer, replicating their design work. This could occur because the programmers lack visibility of the HCI designers work, focusing on the outcome of their analysis (e.g. the prototype). The HCI designer should make their analysis available to programmers so that they can judge which aspects of the HCI design have been well-designed and which aspects of the design just fill space, or complete a story. So, if the HCI designer has worked out how part of the user interface will work at a detailed level, programmers should be guided away from replicating this and should instead be guided towards a more supportive reviewing role.

The duplication of design effort also arises from the programmers not being in the picture when it comes to understanding the current state of the design and the rationale for it. This problem is compounded by the use of the prototype to convey HCI design intent, because prototypes can be ambiguous and do not convey design rationale.

Therefore, the causes of duplicated effort are very similar to those that led to ineffective negotiation between the HCI designer and programmers described in point 4 above.

6. Low Utility of HCI Prototype by Programmers

HCI prototypes that have gradually evolved through the early stages of a project and incorporate early user feedback can provide a concrete means of seeing the HCI design intent for the proposed software. However, if prototypes have been designed to be driven by the person that produced them (e.g. the HCI designer), in a way that follows a particular scenario, they may prove to be inaccessible to anyone else. Thus, prototypes are likely to be under-used by the programmers, even when they have questions about HCI design intent.

Even if programmers do attempt to use these chauffeured prototypes to comprehensively test out the prototype in order to ascertain how the proposed software should work, they may get the wrong impression of HCI design intent. One programmer who took part in the experiment described in chapter 4 had apparently done this:

"I wasn't completely clear how the checklist was supposed to work, not all of the Notes entered were displayed in the log file - maybe this is a bug." (table 4.40, section 4.4.6)

The low utility of prototypes in practice is therefore thought to arise from their inaccessibility and ambiguous nature. This result is not believed to have been previously reported in literature.

5.3 Implications of using ProtoTour in Commercial Practice

This section relates the findings of the experiment that tested the utility of ProtoTour (described in chapter 4) to the likely implications of using it in practice.

1. Using ProtoTour in Practice to Convey an Understanding of HCI Design Intent

The experiment found that use of ProtoTour gave programmers a better understanding of HCI design intent than if they had used a visual prototype and asked a number of questions of the HCI designer. Although the ProtoTour representation used in the experiment was constructed away from usual commercial constraints, it is believed that a comparable representation could be constructed by an HCI designer in a commercial situation. Therefore, it is believed that ProtoTour could genuinely improve programmers' comprehension of HCI design intent in commercial practice.

Although there are no reports of a tool like ProtoTour in the literature, there are a number of findings reported that relate to the visual prototypes, on which ProtoTour is based, that are relevant. Despite few tangible research results being reported, a number of people have claimed that prototypes can help to facilitate communication within a software team (e.g. Damodaran, 1991; Glushko, 1992; Preece *et al.*, 1994; Wagner, 1990; Wilson and Rosenberg, 1988). Other researchers have alluded to the potential of prototyping for facilitating comprehension within a software team (e.g. Gladden, 1982; Heckel, 1991; Miller-Jacobs, 1991; Rudd and Isensee, 1994; Wilson and Rosenberg, 1988). The experiment described in chapter 4 provides evidence that supports the suggestions from literature that prototypes have a use for facilitating communication and comprehension. However, it must be recognised that those recognising this potential are in a minority. This was seen from the fact that a 1995 survey into the use of prototypes (Kinmond, 1995) completely failed to consider this mode of use.

An unexpected finding from the experiment was that ProtoTour apparently reduced the variability in task performance of programmers, compared with their performance using a visual prototype. Concluding an assessment of individual differences among programmers, Curtis (1988) found that the individual differences paradigm had failed to show why differences existed among programmers, or how to reduce them other than through selection. Thus, if ProtoTour could genuinely reduce the variability in performance of programmers, this would be very worthwhile. However, this result from the experiment should be treated cautiously, because the experiment design was not tailored to the analysis of this particular phenomena. Therefore, this result should be considered as a finding worthy of further research. It is believed that there have been no claims from other researchers to have achieved a reduction in the effects of individual differences among programmers through tools or techniques.

2. The Accessibility of ProtoTour in Practice

Experimental results failed to suggest whether ProtoTour would be any more accessible than traditional visual prototypes (i.e. chauffeured prototypes). However, ProtoTour participants were found to have a better understanding of HCI design intent than visual prototype participants, even though they asked less questions of the HCI designer. This perhaps suggests that the information contained in ProtoTour was indeed more accessible, but that this was not measured very effectively in the experiment.

Although the experiment was unable to measure a difference in accessibility of the two approaches, it was clear that the majority of participants from each group (91%) were in favour of using a prototype-centred approach to specifying HCI design intent.

It is believed that the accessibility of ProtoTour would become clear in commercial practice for a number of reasons:

- ProtoTour 'guided tours' of a prototype are designed to be operated by a programmer seeking to understand the HCI design intent for the software they are producing. This has to be inherently more accessible to programmers than chauffeured prototypes designed to be operated by the prototype builder alone, and which do not make much sense outside the context of the walkthrough story;

- ProtoTour should provide programmers with a means of gaining information about HCI design intent without the need to voice what they may fear to be a simple question;

- almost by definition, many programmers prefer an on-line approach to work, which should favour ProtoTour over a written specification in practice.

Ehn (1993) suggested that many aspects of a prototype or mock-up cannot be explicitly described in a formal language. ProtoTour has provided an answer to this by building an on-line explanation around the prototype itself rather than attempting to describe the prototype out of context. It is strongly believed that ProtoTour would constitute a representation of HCI design intent that would be accessible in practice.

3. The Utility of the Extra Information in ProtoTour

Experimental results failed to suggest whether extra information in ProtoTour would help programmers to suggest improved implementation alternatives. A number of the difficulties in the interaction between the HCI designer and programmers centred around the programmers not being in a position to contribute design suggestions on an equal footing with the HCI designer. This was partly because they were not made aware of design rationale for the HCI design presented to them, and partly because their general project understanding was often poor, as they joined the team late. Therefore, the ProtoTour representation aimed to provide programmers with the necessary information to make an effective contribution to the design.

One finding that was apparent from the experiment was that the programmers did not come up with many design alternatives, whether they used ProtoTour or the visual prototype. It is thought that this could be due to 'satisficing' (Simon, 1981). Ball *et al.* (1998) describe this as a tendency to stick with an acceptable design solution with only moderate exploration of the solution space. Thus, it is believed that the apparently polished design solution shown in the ProtoTour or the visual prototype representations can repress further contributions to the design. The 'father of Visual Basic', Alan Cooper has expressed dismay that Visual Basic is so frequently used for early prototyping, precisely because of the danger of becoming fixated with a particular design too early (Cooper, 1994). The practical implication of this very serious problem is that ProtoTour representations must not be used during the early stages of the design process where contributions from programmers are sought. Cooper (1994) suggests that the early stages in the design process should be facilitated by paper-prototyping (a collaborative design technique). This would certainly appear to be an appropriate means of establishing the fundamental features of a design before turning to a visual prototype and subsequently a ProtoTour representation.

A key feature of the extra information in ProtoTour is design rationale. The experiment did not reveal differences in the performance of programmers who had received design rationale information through ProtoTour and those that had to request it of the HCI designer. Other researchers have been unsure whether the benefit of recording design rationale is worth the cost in terms of collection and representation (e.g. Buckingham-Shum, 1996; Grudin, 1996). From this, it can be implied that it is not currently a common feature of software practice. This supposition was supported by findings from the experiment, which illustrated the fact that programmers were unused to receiving such information. However, it is believed that stating the rationale for all aspects of the HCI design has great value in commercial software development. Additional findings from the experiment, described in section 5.2.1, suggest that programmers' perceptions of usability and the role of the HCI designer have scope for improvement, because many seemed to misunderstand the concept of usability. It was apparent that programmers perceived usability as an inherent feature of software, rather than as something that can only be assessed in the context of the users' tasks. Therefore, an additional benefit of conveying design rationale to programmers would be educating them about what usability means. This could contribute to minimising role ambiguity in a software team which includes an HCI designer.

Using a prototype as a common database for project information was recommended by Heckel (1991). Others have suggested that prototypes can be used to answer questions about the design (Wilson and Rosenberg, 1988), and as a means of integrating new team members (Wagner, 1990). Such suggestions from literature do not appear to have been followed up with actual studies of commercial projects utilising prototypes in this way. Although the experiment has not demonstrated the potential of providing extra information alongside an explanation of the prototype, it is believed that this would have merit in commercial software development. In particular, it is thought that this extra information would help the team to maintain a common understanding of the software they are producing and address most of the difficulties in the interaction between the HCI designer and the programmers.

4. Reducing the Volume of Communication Between the HCI Designer and the Programmers

The experiment found that use of ProtoTour could reduce the amount of time that the HCI designer would have to spend explaining aspects of the visual prototype to programmers. In particular, experimental results suggest that ProtoTour would reduce the volume of questions that the HCI designer is likely to asked by programmers about HCI design intent.

From the qualitative investigation described in chapter 2 it is apparent that there is a high volume of communication between the HCI designer and the programmers during the implementation phase of a project. At this stage in a project, the programmers must find out how the software has been designed to work and a major aspect of this is HCI design intent. This kind of interaction between HCI designers and programmers has also been described by Carey *et al.* (1991). They suggested that enquiries programmers made of human factors specialists about a specific system are best supported by context sensitive access to a mock-up. This provides evidence that using prototypes to support explanations of HCI design intent to programmers is a practice that does occur outside the context of the team studied in the qualitative investigation. Although they do not specifically describe problems with the volume of questions directed at the HCI designer by programmers, Carey *et al.*(1991) did suggest that demand for HCI expertise from an in-house resource can soon exceed supply.

The HCI designer in a commercial software team should be aware that they will be faced with a high volume of enquiries from programmers in the implementation stage, which will cause interruptions to their work. If the software project is small, or the team is comprised of highly skilled programmers who work well together, it may be that the volume of enquiries will be low. However, a large, mixed ability software team, perhaps geographically distant from the HCI designer, would provide a clear case for utilisation of ProtoTour to reduce the volume of communication between the HCI designer and the programmers.

5.4 Construction and Deployment of a ProtoTour Representation

A prerequisite for the deployment of ProtoTour is that the software project already has justification for producing a visual prototype, probably for the purposes of eliciting requirements from users. Only in exceptional circumstances might a ProtoTour representation be constructed solely to support the comprehension of HCI design intent. An exceptional circumstance ould be when a small close-knit team of highly skilled designers have produced a design without needing to use a visual prototype (they could have used paper-based methods), but they then need to convey their design to 100 programmers[1]. In such a situation, it may be worth producing a visual prototype solely for the purpose of building a ProtoTour explanation to help convey the HCI design intent. However, in most cases, if a project does not require a visual prototype to be produced for gathering user feedback on the HCI design, it is unlikely to be realistic to produce a ProtoTour representation. The great potential of ProtoTour is that it exploits the visual prototype already constructed for the early phases of design, making use of it throughout the implementation stages.

ProtoTour aims to 'bolt on' to any existing visual prototype regardless of the language used for the prototype development. ProtoTour does not impose any constraints on the production of the visual prototype, because it is important that the flexibility of visual prototyping is not reduced.

The information for a ProtoTour explanation would probably be collected by the HCI designer in the team. The intention is that the HCI designer would not incur much additional work in order to build such a representation. The ProtoTour explanation would gradually evolve alongside the HCI designer's normal design and prototyping activities. ProtoTour aims to be a central repository for the HCI designer's designs, rationale, user profiles, task analysis, scenarios, general project information and any other form of analysis carried out. This kind of design and analysis would have already been carried out by the HCI designer, but the information would be filed away or contained in various reports. ProtoTour aims to collect and organise this information centred around the HCI design shown in the visual prototype – the most concrete view of HCI design intent for much of the project's duration. Thus, ProtoTour does not aim to create more work for the HCI designer – only to provide a central way of storing their work and making it generally available to the team. ProtoTour therefore aims to avoid problems associated with other tools designed to support software processes. For example, Integrated Project Support Environments (IPSEs) and Computer-Aided Software Engineering (CASE) have often failed because of the administration burden they place on the individuals in team (Le Quesne, 1988).

Recording and annotating prototype 'in-use' sequences may take some extra effort on the part of the HCI designer. To produce these, the HCI designer will need to carefully

[1] This is not unlike the situation that occurred during the development of Borland's Quattro Pro for Windows (Gabriel, 1994).

prepare walkthroughs of the prototype and record them (as standard format animation files). These sequences will then need to be edited and annotated or dubbed with an appropriate voice-over. Although this activity will take the HCI designer some time, they should not subsequently need to explain aspects of HCI design intent to programmers – instead programmers should get all the information they need from ProtoTour.

ProtoTour must fit in with the possibly craft-based nature of HCI design (Baeker and Buxton, 1987a; Dowell and Long, 1989; Heckel, 1991; Rubenstein and Hersh, 1987; Wroblewski, 1991), and flexible working practices of the designer. Tools designed to support the software production process that have imposed a particular structure on methods of representing data and processes have failed (e.g. Land et al, 1991). Thus, ProtoTour will provide the HCI designer with a framework designed to accept a broad range of representations using a standard picture format (e.g. .gif) and a standard documentation format (e.g. HTML). This will allow the designer freedom to choose the most appropriate representation or scan in pencil sketches.

The ProtoTour framework will also contain the concept of a topic template. Templates will not be prescriptive but will offer the designer links to various areas where topic related information such as design rationale, cross references to other topics or hock-ups to a walkthrough sequence can be specified. The aim of the template is to help the HCI designer to remember to add any extra information available.

As soon as there is a reasonable amount of content associated with at least one prototype walkthrough, the ProtoTour representation should be made available to the project team so they are kept up-to-date. However, in design practice, satisficing is believed to be a very serious problem - a concern supported by results of the experiment described in chapter 4. This suggests that the deployment of ProtoTour to the team should be resisted until the primary design concepts have been gathered. Showing a polished-looking design in a ProtoTour representation is believed to be a very effective way of stifling radical design ideas from team members, particularly programmers. Such design ideas are of immense value to a software project and could well end up as product differentiators or reduce the development time. Therefore, it is suggested that ProtoTour is deployed when the design team are confident that the key conceptual underpinning of the design is right. Paper-prototyping and collaborative design techniques (Cooper, 1994) could be utilised in the early design stages to get to the point where ProtoTour could be deployed without satisficing becoming a problem.

Web-based intranet has become common in software organisations and would be an ideal vehicle for distributing the ProtoTour representation to the rest of the team in an accessible way. The ProtoTour representation will gradually evolve, so it is important that this is managed in a way that suits the team, so that they are aware of new elements in the ProtoTour and changes to old designs.

5.5 Future Research

This section outlines areas of future research potential derived from the literature review, qualitative investigation, the design and development of ProtoTour, and from the results of the experimental evaluation of the ProtoTour concept.

1. Communication, Comprehension and Conceptual Integrity Within Software Teams

The realisation that producing software is primarily a people problem has not generated enough research attention into aspects of communication and comprehension within software teams. In particular, the importance of maintaining conceptual integrity of the design within the software team is recognised by few and has not received a level of research attention equal with its importance in software production.

2. Integrating HCI and Human Factors Considerations into Software Production

Introducing HCI and Human Factors specialists to software teams is a logical first step towards creating the multi-disciplinary teams that many people advocate. This has been found to be an effective means of introducing HCI design considerations into the heart of software design. However, there is little evidence of such specialists operating within software teams. The rarity of such jobs advertised compared with the abundance of programming jobs is probably the best illustration of this. Future research into the barriers of integrating HCI and Human Factors specialists into software teams would seem relevant.

3. Poor Distribution of General Project Understanding Among Team Members

The distribution of general project understanding among team members was found to be poor in the projects that were the subject of the qualitative investigation. Furthermore, some team members apparently considered there to be no need for them to gain such an understanding, in spite of the day-to-day technical decisions they made. This effect and its implications have not been reported in the literature. Research into these issues would be valuable and may generate a new definition of software developers' roles.

4. Identifying Situations where Recording Design Rationale could be Beneficial

Further research into recording design rationale would appear to be appropriate. The research community appears undecided as to whether the benefit of recording design rationale would outweigh the cost of recording it. Design rationale was not recorded effectively during the projects studied in the qualitative investigation and may have been a partial cause for the volume of enquiries programmers addressed to the HCI designer (hence the inclusion of design rationale in ProtoTour). It would seem appropriate for research to focus more attention on situations where records of design rationale would be beneficial (e.g. perhaps large projects with long duration) and situations where it may have a detrimental effect (e.g. maybe small projects of short duration).

5. Improvements to the Existing ProtoTour Implementation

The construction of the ProtoTour representation used for evaluations in this thesis was complicated by the absence of hypertext authoring tools and animation construction tools. In commercial software practice, it is extremely unlikely that preparation of a ProtoTour representation in this way would be viable.

Tools for the direct authoring of hypertext documents do exist (e.g. RoboHelp by Blue Sky) and these are said to greatly simplify the hypertext authoring task. Authoring aspects of a ProtoTour representation could make use of such hypertext authoring tools to greatly simplify the various explanations.

The recording of scenario animation sequences from the prototype being represented in ProtoTour could also be improved. The DemoQuick toolset used for the example ProtoTour representation made the simple task of recording a Windows application in use very troublesome and lengthy. Other tools with similar functionality to DemoQuick have emerged (e.g. Lotus Guided Tour) and these would be worth investigating.

An integrated ProtoTour tool with hypertext authoring aspects and improved animation capture facilities within the same tool would improve and simplify the linking of hypertext explanations to animated prototype walkthroughs. This would also make the user interface of ProtoTour more seamless.

A further potential improvement to capturing a ProtoTour representation of a prototype would be to enable the HCI designer (or prototype demonstrator) to record a spoken narration of the prototype walkthroughs. This could reduce the burden of constructing a ProtoTour representation by enabling the HCI designer to incidentally record walkthrough animations and spoken narration at the same time. In fact, the HCI designer could make such recordings while demonstrating the existing prototype to members of the software team. This enhancement may substantially increase the viability of producing ProtoTour representations.
The increasingly widespread uptake of web-based intranet, HTML and DHTML provide a strong case for constructing a ProtoTour representation using these technologies. It is strongly recommended that these technologies be considered as a means of improving the ProtoTour concept. Particularly because the internet/intranet/extranet provide a very suitable medium for easily distributing a representation like ProtoTour in a format that is inherently accessible to programmers.

6. Extension of the ProtoTour Concept

Many of the extensions to the ProtoTour concept described below actually formed part of the original ProtoTour concept. They were trimmed from the original concept to retain focus on the specific research question and make evaluation of ProtoTour viable within time constraints.

Recognising that change is inherent in software projects and that continual re-issuing of written specification documents to cope with this is ineffective, even counter-productive, ProtoTour could contain a ‘News’ facility. The News facility would provide a centralised means of conveying all requirements changes and design changes to the software team as they happen. This facility would help to ensure that the conceptual integrity of the software (in the minds of the members of the software team) remains current.

Because ProtoTour is a one-way medium of communication from HCI designer to programmer, a comments window could be added to enable users of the ProtoTour representation to provide feedback. This is something which Wagner (1990) also suggested future prototypes could incorporate (see section 1.11.3.3.2).

One extension to the ProtoTour concept that was considered was to place ProtoTour at the heart of a project support tool. Because, the visual prototype apparently provided the best common currency visualisation of the conceptual model of the software being produced, ProtoTour may provide a good way of organising the implementation effort. If the HCI designer has produced a detailed ProtoTour representation of proposed software, it could then be used to divide the implementation work among the programmers. As a large proportion of programming is now devoted to the user interface (see section 1.2.2), the work of the implementers is often divided by screen elements like Windows. The ProtoTour concept could be extended to support this. For example, after the external design of a proposed software product has been represented in ProtoTour, IT designers could represent the internal design of the proposed software alongside it. Project control features could be built in to enable the project manager to see which elements of the external design have still to be designed internally. They could therefore gain visibility of concrete implementation progress viewed against the conceptualisation provided by ProtoTour. This could further be refined to enable the project manager to monitor the progress of individuals in the software team. Whilst placing ProtoTour at the heart of a project support tool has some intuitive and theoretical appeal, it is felt to be a dangerous and unlikely path for extension of the ProtoTour concept. The reason for this is that such a tool would begin to enforce a structured way of working on the software team. Such enforced procedures and working practices have caused IPSEs and CASE tools to fail to gain acceptance (see section 1.5.1).

If a ProtoTour representation evolves alongside the software product, it could become useful as a project record or archive. From the HCI perspective, it would be useful to review design decisions in the light of summative evaluation of the end product in use. Information from such evaluations often comes too late to have an influence on the actual product. With a ProtoTour archive, the evaluation could be used to improve and revise the heuristics, which HCI designers use in forming new designs. ProtoTour project records could also provide a useful resource library of user interface design concepts. Not only could design ideas be revisited, the rationale could be reviewed as well as evaluation results, demonstrating how well the design concepts fared in practice and maybe indicating how they could be revised in future. From a project management perspective, a ProtoTour archive could be extended to record how long aspects of the software actually took to implement, enabling estimating to be improved. One concern with extending ProtoTour in this direction is that it may not be commercially realistic. Evolving a ProtoTour representation alongside the software product could prove infeasible (unless the evolutionary aspects were relatively minor, such as adding project news items and updates). Possibly even less commercially viable would be the recording of summative evaluations within a ProtoTour archive, because at this stage of the project the budget has usually run out.

7. Further Evaluation of ProtoTour

The experiment conceived and carried out in this thesis was believed to be the best and only viable means of comparing ProtoTour with a traditional prototype. A better comparison may be possible if programmers go to the lengths of actually implementing an aspect of the software illustrated by the ProtoTour representation but this would be extremely expensive and time consuming. An experiment lasting only 1.5 hours cannot produce a faithful simulation of software production practices, which might involve ProtoTour being used for six months or longer. However, the results do provide sufficient incentive to continue to explore the potential of the ProtoTour concept.

The next stage in the evaluation of the ProtoTour is to try it out on a live software project. This could provide information on the utility of the concept and the viability of constructing a ProtoTour representation. However, not only would this kind of evaluation not facilitate any kind of formal comparison with alternative approaches, the actual impact of using ProtoTour in practice would still be hard to assess (see section 4.2.2.2 for a discussion of assessing new techniques in software production). Qualitative assessments of its performance may be the only realistic way to proceed, for example, by asking those programmers participating in the project if the ProtoTour is better than visual prototypes they have used before. There are obviously lots of problems inherent in this kind of comparison - experiential effects, differences in the software projects, differences in the quality of the designs reflected in the prototypes, different individuals in the team, to name but a few. Therefore, a qualitative approach, possibly involving a case study of ProtoTour in action, may be the most realistic means of further evaluating it.

As well as concentrating on utility and viability of ProtoTour in practice, further research is required to discover the project characteristics for which the representation would be best suited. It is likely that ProtoTour would be completely redundant in very small, highly skilled, teams but may be especially useful in larger teams (i.e. more than seven people). It may also be especially useful for situations where the design team and implementation team are completely different and maybe even geographically separated. Such situations usually require a great deal of dependence on written specifications, which are difficult to write and interpret.

8. Why Do Programmers Not Ask About Design Rationale?

One unexpected result from the experiment evaluating ProtoTour was that none of the programmers questioned why the artifact they were working on was designed as it was. Half of the experimental participants were using ProtoTour, which contained an explanation of design rationale, but if programmers had taken an interest in this, it would have probably provoked a discussion. What actually may have occurred is that because the prototype looked polished, it came across to programmers as too concrete and final, leading the programmers to perceive there to be no scope for contributing new ideas (another possible symptom of satisficing). Alternatively, it may be that current practice leads programmers to expect to be given a specification and not question it. If this is the case, it will lead to (or indicates) sub-optimal performance of software teams, particularly multi-disciplinary teams. In a multi-disciplinary team, the person designing the external appearance of the software is unlikely to have technical expertise equivalent to the implementors. This means that implementors are in the best position to suggest alternative implementation approaches, which could equally well meet design criteria but may save considerable implementation time and effort, and could very well provide improvements to the user interface. Clearly, if programmers do not generally consider this as part of their role, sub-optimal performance of software teams will almost be a certainty. Therefore, further research into why programmers do not question why an artifact shown in a prototype is designed as it is needs to be carried out.

9. Programmers' Failure to Suggest Significant Design Alternatives - Satisficing

When specifically asked to suggest alternatives which would speed up the implementation, very few programmers in the evaluation experiment (less than 10%) suggested design alternatives, other than minor changes to the given design. The existence of a 'shiny' prototype may have blocked their ability to disregard the design and come up with something completely different. If prototypes do create this 'block', the consequences could again be sub-optimal team performance (described in section 4.5.4). This kind of 'block' is sometimes referred to as 'satisficing' and has provoked alarmingly little interest in literature relating to software production. A paper by Ball, Maskill and Ormerod (1998) relating to satisficing in engineering design would be a good starting place for an interested reader.

10. Programmers' Failure to Ask Questions About Users and Task

Only one programmer out of 22 in the ProtoTour evaluation experiment asked a question about the user or their tasks. However, as with design rationale, half of the experimental participants were using ProtoTour, which contained some information about users and tasks. As with design rationale, if programmers had taken an interest in this, it would probably have provoked a discussion. In spite of this failure to consider the users and tasks of the software represented in ProtoTour, programmers suggested more design alternatives (albeit mostly minor ones) aimed at improving usability, than speeding up or otherwise improving implementation (see section 4.5.5). This may be partially accounted for by the participants' realisation that the experiment was being carried out by an HCI designer, and they therefore thought that they should be seen to concentrate on usability aspects. However, this is not believed to account for this extent of experimental effect. Programmers apparently believe in generic usability improvements to a user interface based on heuristics and intuition. Although suggesting design alternatives to improve consistency with Windows style conventions would have some validity, even if actual users and tasks were not considered, it was apparent that design alternatives put forward to improve usability were not to do with consistency. This suggests that further research could be conducted to find ways of educating programmers about usability and HCI. Furthermore, programmers concentration on usability aspects of the design was misplaced. This suggests that further research into the scope and responsibilities of roles within a software team would also be beneficial .

11. Can Automated Prototype Walkthroughs Reduce Expectations Problems?

In the evaluation experiment, four programmers in the experimental group using visual prototypes expressed a concern about the issue of increasing user and client expectations which can arise when using visual prototypes (see section 4.5.6). None of the programmers in the ProtoTour experimental group made any comments about this issue. It could be that it is the highly interactive nature of visual prototypes which gives rise to the creation of false and misleading expectations and the myriad of problems this can cause (see section 1.7.5.1). In contrast, the pre-recorded narrated animation sequence, demonstrating exactly the same user interface as seen in the visual prototype but via only a single key operation, has less credibility as 'real software' and therefore does not cause expectations to rise in the same way. Future research in this area intuitively appears to have potential and experimental designs to test the hypothesis are not that hard to conceive. For example, two independent groups of users could be presented with either a chauffeur-driven prototype or a narrated animated sequence. The demonstration of both the prototype and the animation could be identical to facilitate comparison. A post-test questionnaire could be utilised to test participant perceptions of the level of completeness of software, likely delivery timescales, and other measures relating to expectations.

12. Can ProtoTour Reduce the Variability of Programmer Performance?

Results from the experiment described in chapter 4 suggested that ProtoTour reduced the variability of programmer task performance, compared with programmers using a visual prototype (see section 4.5.2 and table 4.44). Researchers into the subject of individual differences among programmers have failed to discover ways of reducing the effects of this phenomena other than through selection (e.g. Curtis, 1988). Therefore, further investigation into why ProtoTour has apparently reduced the effects of individual differences would appear to be worthwhile, particularly as mixed ability teams are believed to be the norm in commercial software development.

References

All citations made in the main body of the thesis appear in this references section and are formatted in accordance with the guidelines of the Human Factors Society (Human Factors Society, 1992). This style of referencing is also in accordance with Cranfield University guidelines (Cranfield Library and Information Service, 1996).

Acosta, R.D., Burns, C.L., Rzepka, E. and Sidoran, J.L. (1994). A Case Study of Applying Rapid Prototyping Techniques in the Requirements Engineering Environment. In Proceedings of the 1st International Conference on Requirements Engineering (pp. 66-73). Colorado Springs, Colo: IEEE.

Agresti, W.W. (1986). What are the new paradigms? In W.W.Agresti (Ed.). New Paradigms for Software Development (Tutorial) (pp. 6-10). Washington, DC : IEEE Computer Society.

Anderson, N.S., and Olson, J.R. (1987). Methods for Designing Software to Fit Human Needs and Capabilities. In R.M. Baeker and W.A.S. Buxton (Eds.) Readings in Human-Computer Interaction: A Multi-disciplinary Approach (pp. 540-554). Los Altos, CA: Morgan Kaufman Publishers.

Axtell, C., Clegg, C., and Waterson, P. (1996). Problems for User Involvement:A Human and Organisational Perspective. In Proceedings of HCI'96 People and Computers XI (pp. 187-200). London, UK: Springer-Verlag.

Bach, J. (1995). Enough About Process: What We Need Are Heroes. IEEE Software,12(2), 96-98.

Baeker, R.M., and Buxton, W.A.S. (1987a). Introduction. In R.M. Baeker and W.A.S. Buxton (Eds.) Readings in Human-Computer Interaction: A Multi-disciplinary Approach (pp. 1-4). Los Altos, CA: Morgan Kaufman Publishers.

Baeker, R.M., and Buxton, W.A.S. (1987b). Case Study A - The Design of a Voice Messaging System. In R.M. Baeker and W.A.S. Buxton (Eds.) Readings in Human-Computer Interaction: A Multi-disciplinary Approach (pp. 5-7). Los Altos, CA: Morgan Kaufman Publishers.

Baker, F.T. (1972). Chief programmer team management of production programming. IBM Systems Journal, 11 (1), 56-73.

Ball, L.J., Maskill, L., and Ormerod, T.C. (1998) Satisficing in engineering design: causes, consequences and implications for design support. Automation in Construction, 7, pp213-227.

Bannon, L.J. and Bødker, S. (1991). Beyond the Interface: Encountering Artifacts in Use. In J. Carroll (Ed.) Designing Interaction: Psychology theory at the human-computer interface (pp. 227-253). New York: Cambridge University Press.

Barley, N (1986). An Innocent Anthropologist – Notes from a Mud Hut. Middlesex, UK: Penguin.

Barnard, P. (1991). The Contributions of Applied Cognitive Psychology to the Study of Human-Computer Interaction. In B. Shackel and S.J. Richardson (Eds.) Human Factors for Informatics Usability (pp. 151-182). Cambridge, UK: Cambridge University Press.

Basili, V.R. and Reiter, R.W. (1981). A controlled experiment quantitatively comparing software development approaches. IEEE Transactions on Software Engineering, 7, 299-320.

Beladay, L.A. and Lehman, M.M. (1979). The characteristics of large systems. In P. Wegner (Ed.), Research Directions in Software Technology (106-138). Cambridge, MA: MIT Press.

Bell, D. and Spencer, E. (1995, April). Ensuring Compatibility in HCI-SE Integration. Presented at the IEE Computing and Control Colloquium on Integrating HCI in the Lifecycle, London, UK.

Bellotti, V.M.E. (1988). Implications of Current Design Practice for the use of HCI Techniques. In Proceedings of HCI'88 - People and Computers IV (pp.13-34). Cambridge, UK: Cambride University Press.

Bellotti, V.M.E. (1990). A Framework for Assessing the Applicability of HCI Techniques in Human-Computer Interaction. In D.Diaper, D.Gilmore, G.Cockton and B.Shackel (eds.) INTERACT'90 (pp.213-218). Amsterdam: Elsevier Science (North Holland).

Bellotti, V.M.E. (1993). Integrating Theoreticians' and Practitioners' Perspectives with Design Rationale. AMODEUS Project Document: Deliverable CP30

Benyan, D.R. (1995, April). Whither the Life Cycle? Presented at the IEE Computing and Control Colloquium on Integrating HCI in the Lifecycle, London, UK.

Billingsley, P.A. (1988). Taking Panes: Issues in the Design of Windowing Systems. In M. Helander (Ed.) Handbook of Human-Computer Interaction (pp. 413-436). Amsterdam: North-Holland.

Blomberg, J., Giacomi, J., Mosher, A. and Swenton-Wall, P. (1993). Ethnographic Field Methods and Their Relation to Design. In D.Schuler and A. Namioka (eds.) Participatory Design: Principles and Practices (pp.123-155). Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Bødker, S. (1991). Through the interface - a human activity approach to user interface design. Hillsdale, NJ: Lawrence Erlbaum Associates.

Bødker, S., Grønbæk, K. and Kyng, M. (1993). Cooperative Design: Techniques and Experiences From the Scandinavian Scene. In D. Schuler and A. Namioka (Eds.) Participatory Design: Principles and Practices (pp.157-175). Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Boddy, W. (1991, December). Specmaster Demonstrator Task Analysis. Unpublished SFK internal report.

Boddy, W. (1992, January). Object-Action Analysis. Unpublished SFK internal report.

Boddy, W. (1992, April). User Interface Rationale. Unpublished SFK internal report.

Boddy, W. (1992a, September). User Interface Design – Specmaster the Discovery. Unpublished SFK internal report.

Boddy, W. (1992b, September). Specmaster Seed User Interface Plan. Unpublished SFK internal report.

Boehm, B. (1981). Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall.

Boehm, B. (1988). A Spiral Model of Software Development and Enhancement. IEEE Computer, 21(5), 61-72.

Boehm-Davis, D.A. (1988). Software Comprehension. In M. Helander (Ed.) Handbook of Human-Computer Interaction (pp. 107-121). Amsterdam: North-Holland.

Branscomb, L.M. (1983). The computer's debt to science. Perspectives in Computing, 3, 4-19.

Brooks, F.P. (1975). The Mythical Man-Month (reproduced in 1995 edition cited)

Brooks, F.P. Jr. (1977). The computer 'scientist' as toolsmith: studies in interactive computer graphics. In Proceedings of the IFIP'77 Conference, (pp. 625-634).

Brooks, F.P. (1986). No silver bullet - essence and accidents of software engineering. In H.J. Kugler (Ed.), Information Processing 86. New York: Elsevier Science Publishers. (reproduced in Brooks, 1995)

Brooks, F.P. (1995). The Mythical Man-Month (20th Anniversary Edition). Reading, MA: Addison-Wesley.

Brooks, R.E. (1980). Studying Programmer Behaviour Experimentally: The Problems of Proper Methodology. Communications of the ACM, 23(4), 207-213.

Brooks, R. (1990). The contribution of practitioner case studies to human-computer interaction science. Interacting with Computers, 2(1), 3-7.

Browne, D.P. (1994). STUDIO - STructured User-interface Design for Interaction Optimisation. NY, NY: Prentice Hall.

Bryman, A. and Burgess, R.G. (1994). Reflections on Qualitative Data Analysis. In A. Bryman and R.G. Burgess (Eds.) Analysing Qualitative Data (pp.216-226). London, UK: Routledge.

Bryman, A. and Burgess, R.G. (1994b). Developments in Qualitative Data Analysis: An Introduction. In A. Bryman and R.G. Burgess (Eds.) Analysing Qualitative Data (pp.1-17). London, UK: Routledge.

Buckingham-Shum, S. and Hammond, N. (1994). Transfering HCI Modelling and DesignTechniques to Practitioners: A Framework and Empirical Work. In G.Cockton, S.W. Draper and G.R.S. Weir (Eds.) People and Computers IX (pp. 313-326). Cambridge, UK: Cambridge University Press.

Buckingham-Shum, S. (1996). Analyzing the Usability of Design Rationale Notation. In T.P. Moran & J.M. Carroll (eds.), Design Rationale: Concepts, techniques, and use (pp. 185-215). Hillsdale, NJ: Lawrence Erlbaum Associates.

Burgess, R.G. (1984). In The Field – An Introduction to Field Research. London, UK: George Allen & Unwin (Publishers) Ltd.

Business Week (1992). Will it sell? Hard to tell, August 17, p. 72.

Button, G. (1994). Occasioned practices in the work of software engineers. In M. Jirotka and J.A. Goguen (Eds.) Requirements Engineering - Social and Technical Issues (pp. 217-240). London, UK: Academic Press.

Carey, T.T., Ellis, M.S. and Rusli, M. (1993). Reusing User Interface Designs: Experiences with a Prototype Tool and High-Level Representations. In Proceedings of the HCI'93 Conference on People and Computers VIII, Tools and Techniques (pp. 203-216). British Informatics Society.

Carey, T., McKerlie, D., Bubie, W. and Wilson, J. (1991) Communicating Human Factors Expertise Through Design Rationales and Scenarios. In D. Diaper and N. Hammond (Eds.), People and Computers VI (pp. 117-130) Cambridge, UK: Cambridge University Press.

Carroll, J.M., Thomas, J.C., and Malhotra, A. (1980). Presentation and representation in design problem solving. British Journal of Psychology, 71, 143-153.

Carroll, J.M., and Rosson, M.B. (1990). Human-Computer Interaction Scenarios as a Design Representation. In Proceedings of HICSS-23: Hawaii International Conference os System Sciences (pp.555-561). IEEE Computer Society Press.

Cockton, G. (1991) Human Factors and Structured Software Development: the Importance of Software Structure. In D. Diaper and N. Hammond (Eds.), People and Computers VI. (pp. 57-72). Cambridge, UK: Cambridge University Press.

Coggman, L. and Cohen, S.J. (1995, April). Usability Engineering from the Suppliers Point of View. Presented at the IEE Computing and Control Colloquium on Integrating HCI in the Lifecycle, London, UK.

Cohill, A.M. (1991). Information Architecture and the Design Process. In J. Karat (Ed.) Taking Software Design Seriously (pp.95-113). London, UK: Academic Press.

Conklin, E.J., and Burgess-Yakemovic, KC. (1996). A Process-Oriented Approach to Design Rationale. In T.P. Moran & J.M. Carroll (Eds.), Design Rationale: Concepts, techniques, and use (pp. 393-427). Hillsdale, NJ: Lawrence Erlbaum Associates.

Cooper, A. (1994) The Perils of Prototyping. Visual Basic Programmers Journal http://www.cooper.com/articles/vbpj_perils_of_prototyping.html (October, 1998)

Cooper, G.E. and Harper, R.P. (1969). The use of Pilot Rating in the Evaluation of Aircraft Handling Qualities (NASA Technical Note 5153). Washington DC

Couger, J.D. and Zawacki, R.A. (1980). Motivating and Managing Computer Personnel. New York: Wiley.

Craig, P.A. (1991). A Graphic Designer’s Perspective. In J. Karat (Ed.) Taking Software Design Seriously (pp.137-155). London: Academic Press.

Cranfield Library and Information Service. (1996). The Prescribed Form for the Presentation of Theses. Cranfield, UK.

Curtis, B. (1981). Substantiating Programmer Variability. Proceedings of the IEEE, 69 (7), 846.

Curtis, B. (1986). By the way, did anyone study any real programmers? In E. Soloway and Iyengar, S. (Eds.) Empirical Studies of Programmers (pp.256-262). Norwood, N.J.: Ablex Pulishing Corp.

Curtis, B. (1988). Five Paradigms in the Psychology of Programming. In M.Helander (Ed.) Handbook of Human-Computer Interaction (pp.87-105). Amsterdam : Elsevier Science Publishers (North Holand).

Curtis, B., Soloway, E.M., Brooks, R.E., Black, J.B., Ehrlich, K., and Ramsey, H.R., (1986). Software Psychology: The need for an Interdisciplinary Program. In Proceedings of the IEEE, 74(8), 1092-1106.

Curtis, B. , Krasner, H., Shen, V. and Iscoe, N. (1987) On Building Software Process Models Under the Lamppost. In Proceedings of the Ninth International Conference on Software Engineering. Silver Spring, MD: IEEE Computer Society, (pp. 96-103).

Damodaran, L. (1991). Towards a Human Factors Strategy for Information Technology Systems. In B. Shackel and S.J. Richardson (eds.) Human Factors for Informatics Usability (pp. 291-324). Cambridge, UK: Cambridge University Press.

Dowell, J., and Long, J. (1989). Towards A Conception For An Engineering Discipline Of Human Factors. Ergonomics, 32 (11), 1513-1535.

Due, B., Jorgensen, A.H. and Nielsen, J. (1991). An Observational Study of User Interface Design Practice. In Proceedings of the Human Factors Society 35th Annual Meeting, System Development: Tools and Methods (pp. 1219-1222). Santa Monica, CA: Human Factors Society.

Eason, K. and Harker, S. (1991). Human Factors Contributions to the Design Process. In B. Shackel and S.J. Richardson (Eds.) Human Factors for Informatics Usability (pp.73-96). Cambridge, UK: Cambridge University Press.

Earthy, J. (1992). HCI, Where's the Practice? In Proceedings of the HCI'92 Conference on People and Computers VII (pp.477-479). Cambridge, UK: Cambridge University Press.

Eberts, R.E. and Brock, J.F. (1988). Computer-Based Instruction. In M. Helander (ed.) Handbook of Human-Computer Interaction (pp. 599-627). Amsterdam: North-Holland.

Ehn, P. (1988). Work-oriented design of computer artifacts. Falköping: Arbetslivscentrum / Almqvist & Wiksell International.

Ehn, P. (1993). Scandinavian Design: On Participation and Skill. In D. Schuler and A. Namioka (Eds.) Participatory Design: Principles and Practices (pp.41-77). Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Elkerton, J, (1988). Online Aiding for Human-Computer Interaction. In M. Helander (Ed.) Handbook of Human-Computer Interaction (pp. 345-364). Amsterdam: North-Holland.

Engelbart, D.C. (1963). A Conceptual Framework for the Aumentation of Man's Intellect. In Howerton and Weeks (Eds.), Vistas in Information Handling Vol. 1 (pp. 1-29). Washington, DC: Spartan Books.

Engelbart, D.C. (1982). Integrated, Evolutionary, Office Automation Systems. In Landau and Bair (Eds.), Emerging Office Systems, Ablex.

Erickson, T.D. (1990). Creativity and Design. In B. Laurel (Ed.) The Art of Human-Computer Interface Design (pp. 1-4). Reading, MA: Addison-Wesley.

Erickson, T. (1995) Notes on design practice: stories and prototypes as catalysts for communication. In J.Carroll (Ed.) Scenario-Based Design: Envisioning Work and Technology in System Development (pp.37-58). New York: Wiley.

Erickson, T. (1996) Design as Storytelling. Interactions, Vol. III, No. 4, 30-35.

Ferrans, J., Males, D., Myhill, C.S., Skilling, K., Titcombe, A.W., Yates, K. (1993, April). Specmaster Seed Base System Software Specification. Version 1.3. Unpublished SFK internal report.

Fitter, M, and Green, T.R.G. (1979). When do diagrams make good computer languages? International Journal or Man-Machine Studies, 11, 235-261.

Fischer, G., Grudin, J., Lemke, A.C., McCall, R., Ostwald, J., Reeves, B., and Shipman, F. (1992). Supporting indirect collaborative design with intergrated knowledge-based design environments. Human-Computer Interaction, 7, 281-314.

Flor, V.F. and Hutchinson, E.L. (1991). Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming During Predictive Software Maintenace. In J. Koenemann-Belliveau, T.G. Moher and S.P.Robertson (eds.), Empirical Studies of Programmers (pp. 36-64). Norwood, NJ: Ablex Publishing.

Gabriel, R.P. (1994). Productivity: Is there a silver bullet?. Journal of Object Oriented Programming, 17 (1) 89-92.

Gilhooly, K. and Green, C. (1997). Protocol Analysis: Theoretical Background. In J.T.E. Richardson (Ed.) Handbook of Qualitative Methods – for Psychology and the Social Sciences (pp.43-54). Leicester, UK: BPS Books

Gladden, G.R. (1982). Stop the lifecycle, I want to get off. ACM Software Engineering Notes, 7(2), 35-39.

Glushko, R. J. (1992). Seven ways to make a hypertext project fail. Technical Communication, 38, 3.

Gomaa, H. (1983). The impact of rapid prototyping on specifying user requirement. ACM SIGSOFT Software Engineering Notes, 8(2), 17-28.

Gordon, V.S. and Bieman, J.M. (1994). Rapid Prototyping: Lessions Learned. IEEE Software, 12(1), 85-95.

Gould, J. (1988). How to design usable systems. In M. Helander (Ed.) Handbook of Human-Computer Interaction (pp. 757-789). Amsterdam: North-Holland.

Gould, J.D., and Boies, S.J. (1987). Human Factors Challenges In Creating a Principle Support Office System - The Speech Filing System Approach. In R.M. Baeker and W.A.S. Buxton (Eds.) Readings in Human-Computer Interaction: A Multi-disciplinary Approach (pp. 25-37). Los Altos, CA: Morgan Kaufman Publishers.

Gould, J.D., and Lewis, C. (1987). Designing for Usability - Key principles and what Designers think. In R.M. Baeker and W.A.S. Buxton (Eds.) Readings in Human-Computer Interaction: A Multi-disciplinary Approach (pp. 528-539). Los Altos, CA: Morgan Kaufman Publishers.

Gould, J.D., Boies, S.J., Levy, S., Richards, J.T, and Schoonard, J., (1990). The 1984 Olympic Message System: A Test of Behavioural Principles of System Design. In J. Preece and L. Keller (Eds.) Human-Computer Interaction - Selected Readings (pp. 260-283). Cambridge, UK: Cambridge University Press.

Green, T.R.G., Davies, S.P., and Gilmore, D.J. (1996). Delivering cognitive psychology to HCI: the problems of common language and to knowledge transfer. Interacting with Computers: the Interdisciplinary Journal of Human-Computer Interaction, 8 (1), 89-111.

Grønbæk, K., Grudin, J., Bødker, S. and Bannon, L. (1993). Achieving Cooperative System Design: Shifting From a Product to a Process Focus. In D. Schuler and A. Namioka (Eds.) Participatory Design: Principles and Practices (pp.79-97). Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Grudin, J. (1988). Why CSCW Applications Fail: Problems in the Design and Evaluation of Organizational Interfaces. Proceedings of the CSCW ‘88, 85-93. New Yorl: ACM. (Reprinted in D.Marca and G.Bock (Eds.), 1992, Groupware: Software for computer-supported cooperative work (pp.552-560). Los Alamitos, CA:IEEE Press.

Grudin, J. (1991). Interactive systems: Bridging the Gaps Between Developers and Users. IEEE Computer, 24(4), 59-69.

Grudin, J. (1993). Obstacles to Participatory Design in Large Product Development Organisations. In D. Schuler and A. Namioka (Eds.) Participatory Design: Principles and Practices (pp.99-119). Hillsdale, New Jersey: Lawrence Erlbaum Associates

Grudin, J. (1996). Evaluating opportunities for design capture. In T.P. Moran and J.M. Carroll (Eds.), Design Rationale: Concepts, techniques, and use (pp. 453-470). Hillsdale, NJ: Lawrence Erlbaum Associates.

Guindon, R. (1990). Designing the design process: Exploting opportunistic thought. Human-Computer Interaction, 5, 305-344.

Grundry, A.J. (1988). Humans, computers, and contracts. In D.M. Jones and R. Winder (Eds.), People and Computers IV (pp. 161-175). Cambridge, UK: Cambridge University Press.

Hakiel, S. (1995, April). A deliverables oriented approach to the integration of HCI into the design and development of software products. Presented at the IEE Computing and Control Colloquium on Integrating HCI in the Lifecycle, London, UK.

Hammersley, M. (1996). Reading Ethnographic Research – A Critical Guide. Harlow, Essex, UK: Addison Wesley.

Hammersley, M. and Atkinson, P. (1983). Ethnography - Principles in Practice. London, UK: Tavistock Publications Ltd.

Hardy, C., Stobart, S., Thompson, B. and Edwards, H. (1995). A Comparison of the Results of Two Surveys on Software Development and the Role of CASE in the UK. In Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering (pp.234-238). IEEE Computer Society Press.

Harker, S. (1991). Requirements Specification and the Role of Prototyping in Current Practice. In J.Karat (Ed.), Taking Software Design Seriously (pp.339-354). London, UK: Academic Press.

Harland, S. (1989). Human Factors Engineering and Interface Development: A Hypertext Tool Aiding Prototyping Activity. In R. McAleese (Ed.) Hypertext - Theory into Practice (pp. 126-137). Norwood, NJ: Ablex Publishing Corp.

Hartley, R. (1980). Computer Assisted Learning. In H.T.Smith and H.R.G. Green (Eds.), Human interaction with computers. London, UK: Academic Press.

Hawkins, J.S. and Reising, J.M. (1983). Is a Picture Worth a Thousand Words - Written or Spoken. In Proceedings of the 27th Annual Meeting of the Human Factors Society (pp. 970-972). Santa Monica, CA: Human Factors Society.

Hays, W.L. (1988) Statistics. Fourth edition. Fort Worth: Holt, Rinehart and Winston.

Heckel, P (1991). The Elements of Friendly Software Design (2nd Edition). San Francisco: Sybex.

Henwood, K.L. (1997). Qualitative Inquiry: Perspectives, Methods and Psychology. In J.T.E. Richardson (Ed.) Handbook of Qualitative Methods – for Psychology and the Social Sciences (pp.25-40). Leicester, UK: BPS Books

Human Factors Society. (1992). Human Factors Authors' Guide. Santa Monica, CA.

Karat, J. and Bennett, J.L. (1991). Using Scenarios in Design Meetings - A Case Study Example. In J. Karat (Ed.) Taking Software Design Seriously (pp.63-94). London: Academic Press.

Karat, J. (1996) User Centred Design: Quality or Quackery? Interactions, 3(4), 18-20.

Kearney, A.T. (1990). Barriers to the Successful Application of Information Technology. Department of Trade and Industry.

Kieras, D.E. (1988). Towards a Practical GOMS Model Methodology for User Interface Design. In M. Helander (ed.) Handbook of Human-Computer Interaction (pp. 135-157). Amsterdam: North-Holland.

Kim, S. (1990). Interdisciplinary Cooperation. In B. Laurel (Ed.) The Art of Human-Computer Interface Design (pp.31-44). Reading, MA: Addison-Wesley.

Kinmond, R.M. (1995). Survey into the Acceptance of Prototyping in Software Development. In Proceedings of the Sixth International Conference on Rapid Systems Prototyping, Shortening the Path from Specification to Prototype (pp.147-152). IEEE Computer Society Press.

Krasner, H. (Ed.) (1986). Proceedings of the Conference on Computer Supported Cooperative Work. Austin, TX: MCC Software Technology Program.

Krippendorf, K. (1980). Content Analysis: An Introduction to Its Methodology. London: Sage.

Land, F., Le Quesne, P., and Wijegunaratne, I. (1991). Implementing systems standards in a merchant bank: An IPSE at Mbank. In K. Legge, C. W. Clegg and N.J. Kemp (Eds.) Case Studies in Information Technology, People and Organisations (pp.175-185). Oxford, UK: NCC Blackwell Limited.

Landis, J.R. and Koch, G.C. (1977). The measurement of observer agreement for categorical data. Biometrics, 33, 159-174.

Lansdale, M.W. and Ormerod, T.C. (1994). Understanding Interfaces - A Handbook of Human-Computer Dialog. London, UK: Academic Press.

Laurel, B. (1990). Introduction. In B. Laurel (Ed.) The Art of Human-Computer Interface Design (pp.xi-xvi). Reading, MA: Addison-Wesley.

Ledermann, W., Hilton, P., Jackson, T.H., Jenkins, C., MacHale, D., Stewart, I., Tall, D.O., Trustrum, K., Unsworth, P.J., Vajda, S., Williams, H.P., and Wylie, S. (1980). Handbook of Applicable Mathematics - Algebra. Vol 1. Chichester, UK: John Wiley.

LeQuesne, P. N. (1988). Individual and Organisational Factors and the Design of IPSEs In The Computer Journal, 31, 391-397

Levi, M.D., and Conrad, F.G. (1996) A Heuristic Evaluation of a World Wide Web Prototype. Interactions, 3 (4), 51-61.

Levy, S. (1995). Insanely Great - The Life and Times of the Macintosh The Computer That Changed Everything. London, UK: Penguin.

Lewis, C., Rieman, J., and Bell, B. (1996). Problem-centred design for expressiveness and facility in a graphical programming system. In T.P. Moran & J.M. Carroll (Eds.), Design Rationale: Concepts, techniques, and use (pp. 147-183). Hillsdale, NJ: Lawrence Erlbaum Associates.

Lim, K.Y., and Long, J. (1994a). Structured Notations to Support Human Factors Specification of Interactive Systems. In G.Cockton, S.W. Draper and G.R.S. Weir (Eds.) People and Computers IX (pp. 313-326). Cambridge: Cambridge University Press.

Lim, K.Y., and Long, J. (1994b). The MUSE Method for Usability Engineering. Cambridge, UK: Cambridge University Press.

Luff, P., Heath, C., and Greatbatch, D., (1994). Work, interaction and technology: The naturalistic analysis of human conduct and requirements analysis. In M. Jirotka and J.A. Goguen (Eds.) Requirements Engineering - Social and Technical Issues (pp. 259-288). London: Academic Press.

McCarthy, J. (1995, Februaury). 23 Rules of Thumb for Shipping Great Software on Time. Presented at the Visual Basic Insiders Technical Summit. London, UK.

McCracken, D.D. and Jackson, M.A. (1981). A Minority dissenting position. In W.W. Cotterman, et al. (Eds.). Systems Analysis and Design - A Foundation for the 80s (pp.551-553). New York: Elsevier.

McGarry, F.E. (1982). What have we learned in the last six years? In Proceedings of the Seventh Annual Software Engineering Workshop (SEL-82-007). Greenbelt, MD: NASA Goddard Space Flight Center.

McKerlie, D., MacLean, A. (1993). Experience with QOC Design Rationale. In Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems Adjunct Proceedings (p. 519). ACM.

MacLean, A., Young, R.M., Bellotti, V.M.E., and Moran, T.P. (1991). Design Space Analysis: Bridging from Theory to Practice Via Design Rationale. In proceedings of ESPRIT Conference (pp.720-730)

MacLean, A., Young, R.M., Bellotti, V.M.E., and Moran, T.P. (1996). Questions, options and criteria: Elements of design space analysis. In T.P. Moran & J.M. Carroll (Eds.), Design Rationale: Concepts, techniques, and use (pp. 53-105). Hillsdale, NJ: Lawrence Erlbaum Associates.

Mantei, M.M., and Teorey, T.J. (1988). Cost / Benefit Analysis for Incorporating Human Factors in the Software Lifecycle. Communications of the ACM, 31, 428-439.

Martin, C.F. (1988). User-Centred Requirements Analysis. Englewood Cliffs, NJ: Prentice-Hall.

Mason, J. (1994). Linking Qualitative with Quantitative Data Analysis. In A. Bryman and R.G. Burgess (Eds.) Analysing Qualitative Data (pp.89-110). London, UK: Routledge.

Mason, R.E.A. and Carey, T.T. (1983) Prototyping interactive information systems. Communications of the ACM, 26(5),. 347-354.

Maxwell, K.J. (1996, January). A System Performance-Based Approach to Integrating Software Engineering and Human-Computer Interaction Requirements. Presented at the joint British Computer Socienty Requirements Engineering Group and British Human Computer Interaction Group User-Centred Requirements Engineering Workshop: Integrating Methods from Software Engineering and Human-Computer Interaction, York: UK.

Mayes, J.T., Draper, S.W., McGregor, A.M., and Oatly, K. (1990). Information Flow in a User Interface: The Effect of Experience and Context on the Recall of MacWrite Screens. In J. Preece and L. Keller (Eds.) Human-Computer Interaction - Selected Readings (pp. 222-234). Cambridge: Cambridge University Press.

Miles, M.B. and Huberman, M. (1984). Qualitative Data Analysis. Beverly Hills, CA: Sage.

Mills, H.D. (1971). Chief Programmer Teams: Principles and Procedures (Tech. Rep. IBM-FSC 71-5108). Gaitherburg, MD: IBM Federal Systems Division.

Miller-Jacobs, H.H. (1991). Rapid Prototyping: An Effective Technique for System Development. In J. Karat (ed.), Taking Software Design Seriously (pp. 273-286). London: Academic Press.

Moran, T. and Carroll, J.M. (1996). Overview of Design Rationale. In T.P. Moran and J.M. Carroll (Eds.) Design Rationale - Concepts, Techniques, and Use (pp. 1-19). Mahwah, NJ: Lawrence Erlbaum Associates.

Morgan, M. (1996). Qualitative Research: a package deal. The Psychologist, 9(1), 31-32.

Mountfield, S. J. (1990). Tools and Techniques for Creative Design. In B. Laurel (ed.) The Art of Human-Computer Interface Design (pp.17-30). Reading, MA: Addison-Wesley.

Muller, M. (1993). PICTIVE: Democratizing the Dynamics of the Design Session. In D. Schuler and A. Namioka (Eds.) Participatory Design: Principles and Practices (pp.211-237). Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Myers, B.A. and Rosson, M.B. (1992). Survery on User Interface Programming. In P. Bauersfeld, J. Bennett and G. Lynch (Eds.), Procedings of ACM Conference on Human Factors in Computing Systems. Monterey, CA: ACM Press.

Myhill, C.S. (1992, September). The Specmaster Seed System – Task Analysis. Draft 1.0. Unpublished SFK internal report.

Myhill, C.S. (1992, October). The Specmaster Seed System – Non-Functional Requirements. Draft 2.0. Unpublished SFK internal report.

Myhill, C.S. (1992, November). The Specmaster Seed System – Object Model. Draft 4.0. Unpublished SFK internal report.

Myhill, C.S. (1993, September). Human-Computer Interaction for complex, multi-user integrated engineering environments. Unpublished Postgraduate Training Partnership scheme report.

Myhill, C.S. (1993, October). HHI in GUI, OK? Presented at British HCI Group Seminar on Designing for Clear Communication, London, UK.

Myhill, C.S. (1994, August). A User-Centred Development Framework Facilitating Multi-Disciplinary Collaboration for Commercial Software Development. Unpublished Postgraduate Training Partnership scheme report.

Myhill, C.S. (1995, May). Communicating Human-Computer Interaction Design Intent Within a Software Team. Unpublished Postgraduate Training Partnership scheme report.

Myhill, C.S. and Brooks, P. (1996). Communicating Human-Computer Interaction Design Intent: Requirements for Recycling Throwaway Prototypes. In Proceedings of the First International Conference on Engineering Psychology and Cognitive Ergonomics. (in press).

Myhill, C.S., Cocker, S., and Brooks, P. (1994). A Practical Prototyping Process for Human-Centred Software Development: When, Why and How to Prototype. In Ancillary Proceedings of HCI'94 - People and Computers IX. British Computer Society.

Nelson, T.H. (1990). The Right Way to Think About Software Design. In B. Laurel (Ed.), The Art of Human-Computer Interface Design (pp.235-243). Reading, MA: Addison-Wesley.

Newman, W. (1991). Interface Design Issues for the System Designer. In B. Shackel and S.J. Richardson (Eds.) Human Factors for Informatics Usability (pp. 121-131). Cambridge, UK: Cambridge University Press.

Nielsen, J. (1992). Finding usability problems through heuristic evaluation. In Procedings of the ACM Conference on Human Factors in Computing Systems (pp. 373-380). Monterey, CA: ACM Press.

Nielsen, J. (1993). Usability Engineering. Boston: Academic Press.

Norman, D. (1986). Cognitive Engineering. In D.A. Norman and S.W. Draper (Eds.) User Centred System Design - New Perspectives on Human-Computer Interaction (pp. 31-61). Hillsdale, NJ: Lawrence Erlbaum Associates.

Norman, D.A. (1987). Some Observations on Mental Models. In R.M. Baeker and W.A.S. Buxton (Eds.) Readings in Human-Computer Interaction: A Multi-disciplinary Approach (pp. 241-244). Los Altos, CA: Morgan Kaufman Publishers.

Norman, D.A. (1988). The Psychology of Everyday Things. New York, NY: Basic Books.

Norman, D.A. (1990). Why Interfaces Don't Work. In B. Laurel (Ed.), The Art of Human-Computer Interface Design (pp.209-219). Reading, MA: Addison-Wesley.

Norman, D.A. (1996) Interview: A Conversation with Don Norman. Interactions Online. http://www.acm.org/interactions/vol2no2/interview/dnorman (4th February, 1997)

Norušis, M.J. (1993). SPSS for Windows Base System User's Guide Release 6.0. Chicago, Illinois: SPSS Inc.

Olson Jr, D.R., Buxton, W., Ehrich, R., Kasik, D.J., Rhyne, J.R., and Sibert, J. (1990). A context for User Interface Management. In J. Preece and L. Keller (Eds.) Human-Computer Interaction - Selected Readings (pp. 402-420). Cambridge, UK: Cambridge University Press.

Overmyer, S.P. (1991). Revolutionary vs. Evolutionary Rapid Prototyping: Balancing Software Productivity and HCI Design Concerns. In H.-J. Bullinger (Ed.) Human Aspects in Computing: Design and Use of Interactive Systems and Work with Terminals (pp.303-307). Amsterdam: Elsevier Science Pulications.

Perlman, G. (1988). Software Tools for User Interface Development. In M. Helander (Ed.) Handbook of Human-Computer Interaction (pp. 819-833). Amsterdam: North-Holland.

Perzylo, L. (1993). The application of multimedia CD-Roms in schools. British Journal of Educational Technology, 24 (3), 191-199.

Pidgeon, N. (1997). Grounded Theory: Theoretical Background. In J.T.E. Richardson (Ed.) Handbook of Qualitative Methods – for Psychology and the Social Sciences (pp.75-85). Leicester, UK: BPS Books

Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S. and Carey, T. (1994). Human-Computer Interaction. Workingham, England: Addison-Wesley.

Rachel, J. (1997). Ethnography: Practical Implementation. In J.T.E. Richardson (Ed.) Handbook of Qualitative Methods – for Psychology and the Social Sciences (pp.113-125). Leicester, UK: BPS Books.

Rheingold, H. (1990). An Interview with Don Norman. In B. Laurel (ed.) The Art of Human-Computer Interface Design (pp.5-10). Reading, MA: Addison-Wesley.

Richards, L. and Richards, T. (1991) Computing and Qualitative Analysis: Computational Paradigms and Research Processes. In N.G. Fielding and R.M. Lee (Eds.) Using Computers in Qualitative Research. London: Sage.

Ritchie, J. and Spencer, L. (1994). Qualitative Data Analysis for Applied Policy Research. In A. Bryman and R.G. Burgess (Eds.) Analysing Qualitative Data (pp.173-194). London, UK: Routledge.

Rittel, H.W.J., and Weber, M.M. (1973). Dilemas in general theory of planning. Policy Sciences, 4, 155-169.

Robson, C. (1998) Real World Research: A Resource for Social Scientists and Practitioner-Researchers. Oxford, UK: Basil Blackwell.

Rubenstein, R., and Hersh, H. (1987). Design Philosophy. In R.M. Baeker and W.A.S.Buxton (Eds.) Readings in Human-Computer Interaction: A Multi-disciplinary Approach (pp. 502-507). Los Altos, CA: Morgan Kaufman Publishers.

Rudd, J. and Isensee, S. (1994). Twenty-Two Tips for a Happier, Healthier Prototype. Interactions, 1(1), 35-40

Sackman, H. (1970). Man-Computer Problem Solving. New York:Auerbach.

Schutt, R.K. (1996). Investigating the Social World – The Process and Practice of Research. Thousand Oaks, CA: Pine Forge Press.

Scott, R.F. and Simmons, D.B. (1975). Predicting programming group productivity: A communications model. IEEE Transactions on Software Engineering, 1 (4), 411-414.

Scuprowicz, B. (1990). Future Technology: a huge boom ahead for interactive multimedia. Computer World July 20th.

Shackel, B, (1991). Usability - Context, Framework, Definition, Design and Evaluation. In B. Shackel and S.J. Richardson (Eds.) Human Factors for Informatics Usability (pp. 21-38). Cambridge, UK: Cambridge University Press.

Sheil, B.A., (1987). The Psychological Study of Programming. In R.M. Baeker and W.A.S. Buxton (Eds.) Readings in Human-Computer Interaction: A Multi-disciplinary Approach (pp. 165-174). Los Altos, CA: Morgan Kaufman Publishers.

Simon, H.A. (1981). The Sciences of the Artificial. 2nd Edition. Cambridge, MA: MIT Press.

Spradley, J.P. (1980). Participant Observation. New York: Holt, Rinehart and Winston

Stake, R.E. (1994). Case Studies. In N.K. Denzin & Y.S.Lincoln (Eds.), Handbook of Qualitative Research (pp. 236-247). Thousand Oaks, CA: Sage.

Stake, R.E. (1995). The Art of Case Study Research. Thousand Oaks, CA: Sage.

Stapleton, J. (1996, February). Quality Management of RAD. Presented at a British Computer Society Bedford branch and British Computer Society Quality SIG meeting. Cambridge, UK.

Stephens, M (1996). Madness in the Method. D3 - Directions in Desktop Development, October 1996, 34-36.

Stewart, T. (1991). Helping the I.T. Designer to Use Human Factors. In B. Shackel and S.J. Richardson (Eds.) Human Factors for Informatics Usability (pp. 97-119). Cambridge, UK: Cambridge University Press.

Sutcliffe, A. (1992). HCI, Where's the Practice? Position statement made at the HCI, Where's the Practice? panel session in Proceedings of the HCI'92 Conference on People and Computers VII (pp.477-479). Cambridge, UK: Cambridge University Press.

Swartout, W. and Balzar, R. (1982). On the inevitable intertwining of specification and implementation. Communications of the ACM, 25, 438-440.

Tabachnick, B.G. and Fidell, L.S. (1983). Using Multivariate Statistics. New York, NY: Harper and Row.

Thielen, D. (1991). Unconventional Thoughts on Managing a PC Software Development. Microsoft Developers' CD- ROM version 5.

Toren, C. (1997). Ethnography: Theoretical Background. In J.T.E. Richardson (Ed.) Handbook of Qualitative Methods – for Psychology and the Social Sciences (pp.102-112). Leicester, UK: BPS Books

Trenner, L. (1995, April). Prototyping for Usability: Benefits of Peer Group Review. Presented at the IEE Computing and Control Colloquium on Integrating HCI in the Lifecycle, London, UK.

Turner, B.A. (1994). Patterns of Crisis Behaviour: A Qualititative Inquiry. In A. Bryman and R.G. Burgess (Eds.) Analysing Qualitative Data (pp.195-215). London, UK: Routledge.

Tyldesley, D. A. (1990). Employing Usability Engineering in the Development of Office Products. In J. Preece and L. Keller (Eds.) Human-Computer Interaction - Selected Readings (pp. 284-295). Cambridge, UK: Cambridge University Press.

Vertelney, L. and Booker, S. (1990). Designing the Whole Product User Interface. In B. Laurel (Ed.) The Art of Human-Computer Interface Design (pp.57-63). Reading, MA: Addison-Wesley.

Waern, K-G. (1988). Cognitive Aspects of Computer Aided Design. In M. Helander (Ed.) Handbook of Human-Computer Interaction (pp. 701-708). Amsterdam: North-Holland.

Wagner, A. (1990). Prototyping: A Day in the Life of an Interface Designer. In B. Laurel (Ed.) The Art of Human-Computer Interface Design (pp.79-84). Reading, MA: Addison-Wesley.

Wasserman, A.I. (1987). Extending State Transition Diagrams for the Specification of Human-Computer Interaction. In R.M. Baeker and W.A.S. Buxton (Eds.) Readings in Human-Computer Interaction: A Multi-disciplinary Approach (pp. 561-575). Los Altos, CA: Morgan Kaufman Publishers.

Wasserman, A.I. and Shewmake, D.T. (1990). The Role of Prototypes in the User Software Engineering (USE) Methodology. In J. Preece and L. Keller (Eds.) Human-Computer Interaction - Selected Readings (pp. 385-401). Cambridge, UK: Cambridge University Press.

Weinberg, G.M. (1971). The Psychology of Computer Programming. New York: Van Nostrand Reinhold.

Whiteside, J., Bennett, J. and Holtzblatt, K. (1988). Usability Engineering: Our Experience and Evolution. In M. Helander (Ed.) Handbook of Human-Computer Interaction (pp. 791-817). Amsterdam: North-Holland

Wilson, J. and Rosenberg, D. (1988). Rapid Prototyping for User Interface Design. In M. Helander (Ed.) Handbook of Human-Computer Interaction (pp. 859-875). Amsterdam: North-Holland.

Wilson, S., Bekker, M., Johnson, H. and Johnson, P. (1996). Costs and Benefits of User Involvement in Design: Practitioner's Views. In Proceedings of HCI'96 People and Computers XI (pp. 221-240). London, UK: Springer-Verlag.

Woods, D.D. and Roth, E.M. (1990). Cognitive Systems Engineering. In M. Helander (Ed.) Handbook of Human-Computer Interaction (pp. 3-43). Amsterdam: North-Holland.

Wroblewski, D.A. (1991). The Construction of Human-Computer Interfaces Considered as a Craft. In J. Karat (ed.), Taking Software Design Seriously (pp. 1- 19). London, UK: Academic Press.

Yin, R. K. (1994) Case Study Research: Design and Methods. Second Edition. Thousand Oaks: Sage Publications.

Zachary, W.W. (1988). Decision Support System: Designing to Extend the Cognitive Limits. In M. Helander (Ed.) Handbook of Human-Computer Interaction (pp. 997-1030). Amsterdam: North-Holland.

Appendix A Chronological Narrative Description of Project 1

Introduction

This appendix comprises a first person narrative description of the Project 1 software development from the perspective of the participant-observer.

Where possible members of the software team have been coded to protect their identities throughout this narrative. The coding strategy relates primarily to the technical core of the team where more than one individual performed a similar role. Coding cannot effectively protect the identities of team specialists who solely perform a role, such as the project manager. The coding strategy has been set out as follows:

- Software Designer / Programmers are coded SD/P_1, SD/P_2, SD/P_3,
- Programmers are coded P_1, P_2, P_3.

September (1992)

On joining SfK I was immediately assigned to Project 1 in order to make best use of the one week handover period from the person I was replacing. My predecessor was a technical author with a particular aptitude for graphics, so had been involved in designing the user interface of the Project 1 Demonstrator. Before leaving, she wrote a report on the process used to implement user interface designs on the Project 1 Demonstrator and a plan for defining and implementing the user interface for the full Project 1 software product. The main stages in the plan can be seen in figure A.1.

i. **Investigation**
Domain and problem familiarisation.

ii. **Material and Examples**
Analyse sample material and forms used to specify valves from engineers in industry.

iii. **Non-Functional Requirements**
State main objectives of software, user population and requirements and look and feel intent.

iv. **Task Model**
Document activities involved in the current task of specifying control valves.

v. **Object Model**
Conversion of task model to a conceptual model of the software by identifying key objects that are required to be in the software and relationships between objects.

vi. **System Map**
Overview of the Object Model showing key objects and interactions

vii. **Screen mock-ups**
Storyboard of the basic system walk-through, followed by detailed designs of interaction segments, e.g. the login sequence.

viii. **Windows Design Guidelines**
Specification of some basic style guidelines that the development should adhere to.

ix. **User Interface Rationale**

Figure A.1 Plan for Defining and Implementing the User Interface of the Project 1 Software

As well as the plan, my predecessor had also carefully documented the activities involved in designing and implementing the Project 1 Demonstrator, and compiled relevant Visit Reports[1] (most of which related to knowledge elicitation sessions with control valve experts at ICI, i.e. domain related information).

As a new employee, new team member and an HCI designer for the first time, my initial task was familiarisation, which corresponded to ***investigation*** on the plan. My desk was situated between SD/P_1, the senior software designer/programmer, a quiet and shy

[1] a Visit Report is a document produced following some kind of client/expert/user contact and it is designed to capture and convey key information discovered.

person in his early forties who had been programming for over 20 years, and a louder, more dynamic knowledge engineer in his mid twenties. I was ultimately to work with both of my neighbours but to begin with was left to familiarise myself with the project alone.

Before my predecessor left she stressed that I should look carefully at Microsoft Windows[2] and other applications running under Windows, as part of the ***investigation*** activity. As I was already a Windows user I initially felt this to be unnecessary but soon learned that it was an extremely complicated environment which needed careful consideration in order to design compatible Windows applications. Therefore, one dimension of the familiarisation task was to take a deeper look at Windows and Windows applications. Another aspect of familiarisation was to gain an understanding of the domain, which involved learning about control valves and processes surrounding their specification and selection. This task took the form of talking with in-house experts on the subject, looking at actual control valves in BHRG's (SFK's hydromechanics parent company) laboratory and reading engineering texts and training material on control valves. A further aspect of familiarisation was an investigation into the history of Project 1. This took the form of reading the original Scope study which was by now a year out of date, and Visit Reports which were current but did not formally specify requirements for the Project 1 software. The main reference and most up to date document at the outset of the project was the Feasibility document, which although fairly sketchy, did convey the basic intent of the venture.

A key part of the investigation phase to me as a new HCI designer, was to try to work out what exactly my role on the project would be. I began this in earnest by reviewing the documentation produced by my predecessor from the development of the Project 1 Demonstrator. The available documentation was categorised according to the interface development plan introduced in figure A.1. Much of the documentation was extremely hard to follow in that it was often unclear what the inputs to each document were, how to follow the analysis, and where the output fed in to another stage. Table A.1 is a summary of the user interface related documentation of the Project 1 Demonstrator, showing perceived inputs, analysis and outputs of each stage. The table encompasses results of my examination of the documentation, in terms of what I perceived as inputs to each activity, analysis carried out in the activity itself and the output produced.

[2] hereafter abbreviated to Windows

Table A.1. Summary of the User Interface Related Documentation of the Project 1 Demonstrator, Showing Perceived Inputs, Analysis and Outputs of Each Stage

User Interface Task	Inputs	Analysis	Outputs
Investigation	Scope Study, Visit Reports, discussion with colleagues (esp. knowledge engineers)	Personal - intended to improve designer's own knowledge of and feeling for the problem	Improved designer's knowledge of and feeling for problem and domain
Material & Examples	Collection of forms used in industry for specifying control valves	Analysis of actual completed forms, industry guidelines and symbols	Design notes and issues raised, (e.g. red triangle notation)
Non-Functional Requirements	Designers understanding, project objectives	Collate information (e.g. identifying users)	Non-Functional Requirements document
Task Model	Scope study, understanding of designer and knowledge engineer	Analysis of current user task in a hierarchial format	Graphical hierarchial Task Model without supporting documentation
Object Model	Task Model, designer's knowledge, previous project documentation	Highlighting key conceptual objects within the software and their interrelationships	Object Model with little supporting explanation
System Map	Object model, designer's knowledge	Producing a simple overview of the object model	System Map overview of the software
Screen Mock-ups	Object Model, System Map and designer's knowledge	Storyboard sketches of main paths through software, detailed design of conceptual chunks (e.g. the login screen)	Screen Sketches and storyboard
Windows Design Guidelines	Designer's knowledge of Windows style and personal style preference	Looking at Windows applications, identifying key features which must have a visual style specified	Style Guidelines
User Interface Rationale	Designer's knowledge of design decisions	Post hoc record of design decisions made and justification for specific user interface effort	Rationale document

Following the review of the documentation produced for the Demonstrator the next task was to produce a comparable set of documentation for the Project 1 software. It soon became apparent that, using the Demonstrator documentation as a starting point, there was little new concrete information available to add. In the period following the construction of the Demonstrator up until the start of Project 1, the majority of the information gathered had been by knowledge engineers on knowledge elicitation visits to ICI, and by in-house salesman visiting potential customers. At this time there began to emerge many different, often conflicting and mainly informal requirements for the Project 1 software. What this meant to me as an HCI designer was that I could add nothing to the ***Investigation*** stage except to improve my own understanding of the problem.

The ***Material and Examples*** stage involved reviewing engineering documents collected for the Demonstrator project and supplementing this information with a little research into British and American Control Valve specification standards (largely ignored by industry). As with the *Investigation* stage, my major contribution was improving my own knowledge.

Revising the ***Task Model*** for the Project 1 software was my first major contribution to the project. The ***Task Model*** for the Demonstrator consisted of a hierarchial task description of the process of specifying control valves, in diagrammatic format with little supporting explanation. This model had been based on a *Process Plant Design Lifecycle* diagram in the Scope Study and from the combined knowledge of in-house knowledge engineers and technical experts. This document proved to be my first big stumbling block as the diagrammatic format of the ***Task Model*** did not provide enough of a sense of the task. As I had no access to users and no way of formally analysing the task for myself, I had to rely on conversations with colleagues to comprehend and ratify the model. The main addition that I made to the Demonstrator documentation for this early stage of task analysis was to convert the existing hierarchial diagram into a more conventional Hierarchial Task Analysis (HTA) as may be seen in figure A.2. Whilst I considered several available techniques and representations available in literature, HTA seemed to provide the most realistic approach to my situation, i.e. having no formal way to analyse a task and no actual users to analyse. HTA enabled me to make use of the structure of the existing diagrams but also add plans to the diagram to indicate the sequence of tasks, dependencies and to show basic logic, e.g. IF task 1 AND task 2 THEN task 3. This provided more information to the reader of the diagrams but I still felt this likely to be insufficient for some team members, so I also produced written explanation of the tasks and plans. At a later stage, when much of the conceptual design of the Project 1 software had been carried out, there was a requirement for a further task analysis activity. A model of how the user tasks would look after the Project 1 software had been implemented was produced and dubbed the ***Future Task Model*** and will be described in more detail later.

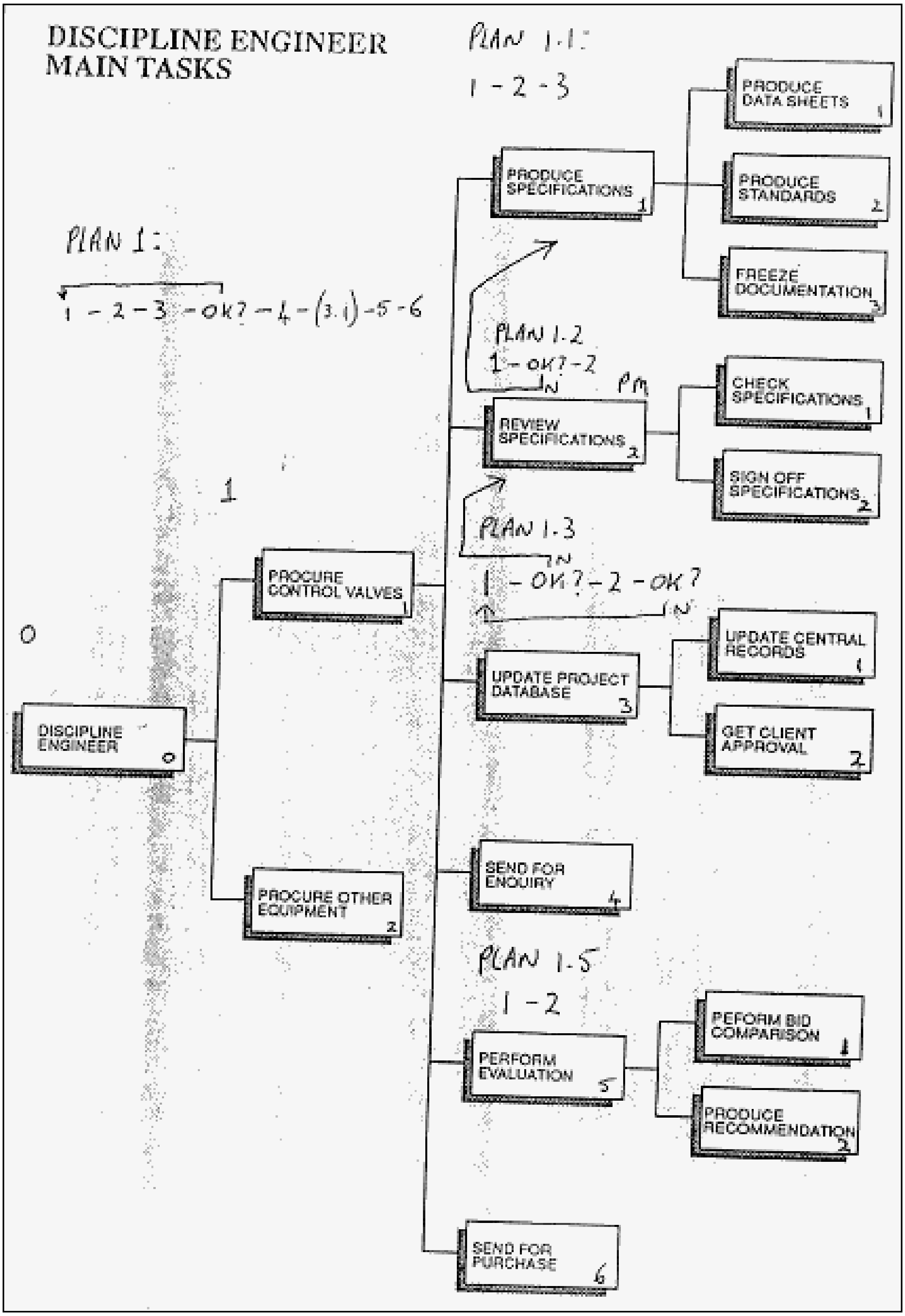


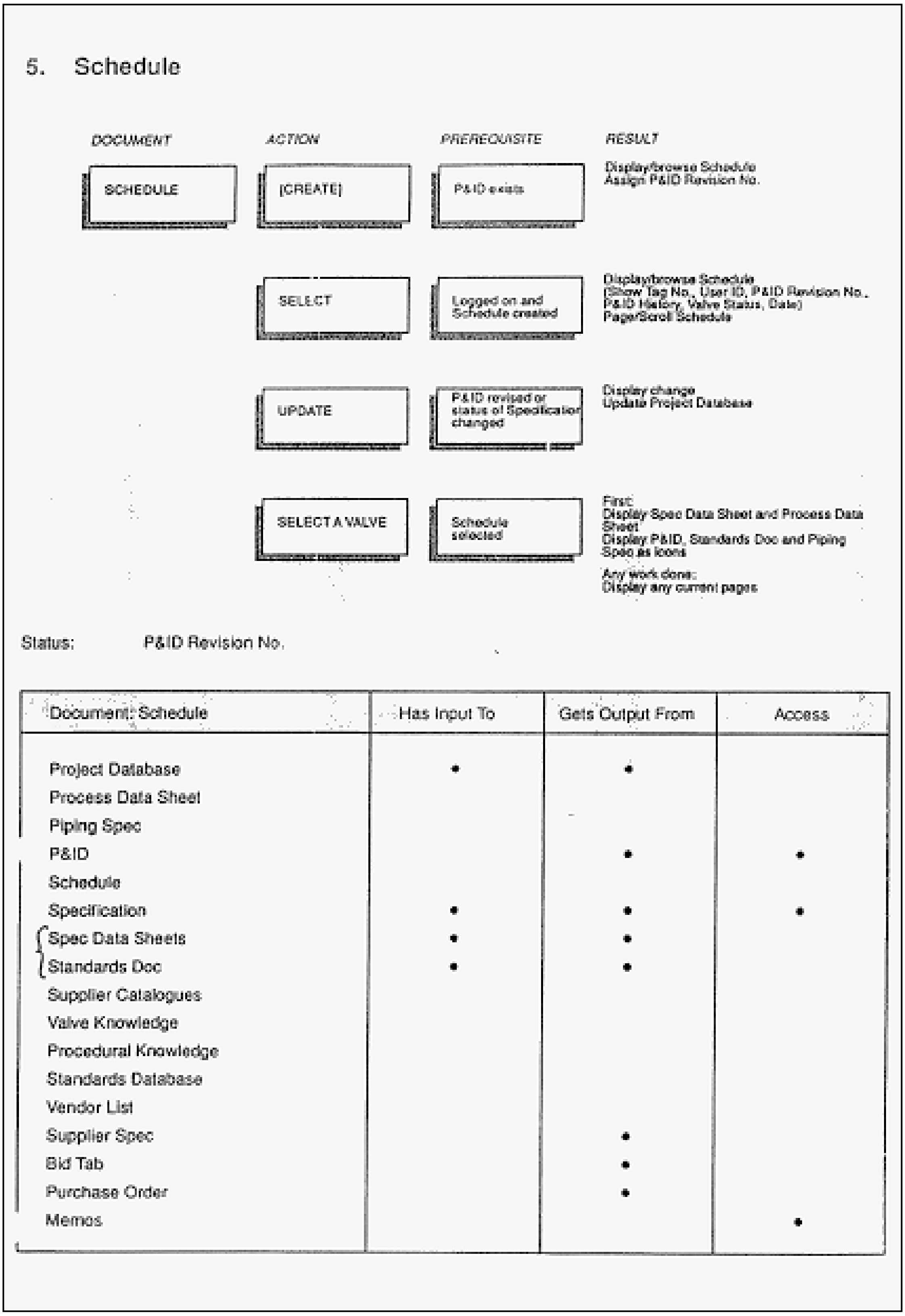
Figure A.2 Work Sample Showing HTA Diagrams with Plans Added

October 1992

Taking the Demonstrator document as a starting point, I re-wrote the ***Non-Functional Requirements*** to highlight the environmental and user considerations that the software should address. Ultimately, I believed this document to be more of an exercise that I had been set by management to ensure that I had a good grasp of the problem. The document expressed my understanding of the objectives of the software, the target user companies, the structure of process plant companies, licensed processes, the proposed end-users and the software requirements. A particular difficulty with this document was representing what was known about the proposed end-users in a meaningful way that would be of use during the design process. The Non-Functional requirements document included generalisations about the level of education and skills typical of a mechanical engineer, but these were informed guesses rather than formally derived. The information used to produce the document came from the picture I had built up of the domain and the proposed solution, through talking with technical management, the salesman, knowledge engineers and by analysing documents such as the Feasibility report. At this stage it became clear that there were differences in opinion between the technical and sales people, about what the software would do. The utility of the ***Non-Functional Requirements*** document was very low and it was not widely studied by members of the project team. The main benefit of producing the document from my viewpoint, was that the process of researching and documenting what we collectively knew enabled me to learn more about the problem. Thus, my increased knowledge of the problem was the main benefit of this activity rather than the document produced.

November 1992

Following on from the ***Task Modelling*** activity and with the deeper knowledge of the problem I had gained from documenting the ***Non-Functional Requirements***, the next step was to extend the ***Object Model*** for the Demonstrator to an ***Object Model*** for the Project 1 software. Again, as with the task analysis, the predominantly diagrammatic format of the ***Object Model*** in the Demonstrator documentation made it extremely difficult to understand the information presented, as may be apparent from work sample shown in figure A.3.



5. Schedule

DOCUMENT	ACTION	PREREQUISITE	RESULT
SCHEDULE	[CREATE]	P&ID exists	Display/browse Schedule Assign P&ID Revision No.
	SELECT	Logged on and Schedule created	Display/browse Schedule (Show Tag No., User ID, P&ID Revision No., P&ID History, Valve Status, Date) Page/Scroll Schedule
	UPDATE	P&ID revised or status of Specification changed	Display change Update Project Database
	SELECT A VALVE	Schedule selected	First: Display Spec Data Sheet and Process Data Sheet Display P&ID, Standards Doc and Piping Spec as icons Any work done: Display any current pages

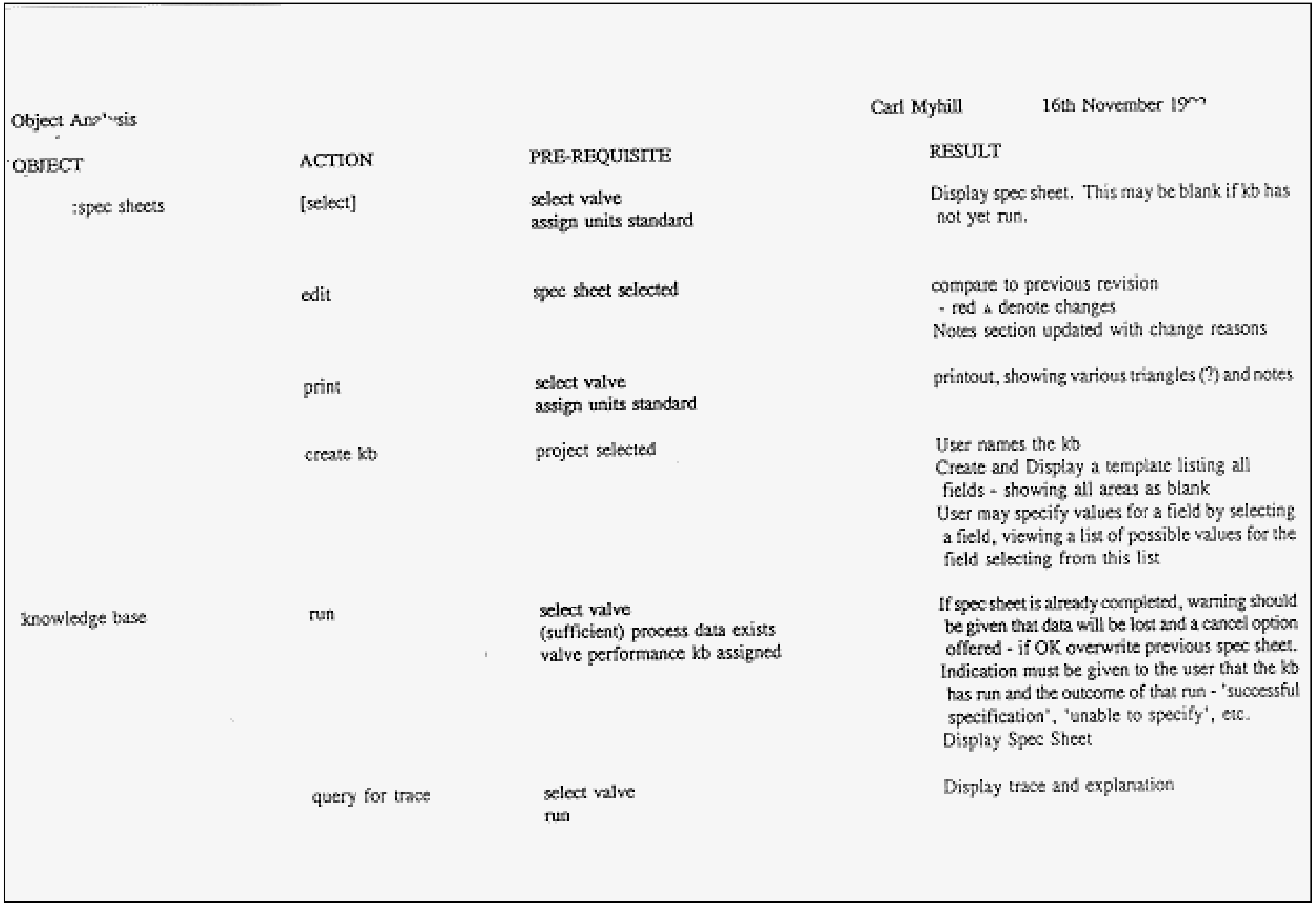
Status: P&ID Revision No.

Document: Schedule	Has Input To	Gets Output From	Access
Project Database	•	•	
Process Data Sheet			
Piping Spec			
P&ID		•	•
Schedule			
Specification	•	•	•
{Spec Data Sheets	•	•	
{Standards Doc	•	•	
Supplier Catalogues			
Valve Knowledge			
Procedural Knowledge			
Standards Database			
Vendor List			
Supplier Spec		•	
Bid Tab		•	
Purchase Order		•	
Memos			•

Figure A.3. Work Sample of an Object Model Segment Centred on the Project 1 Demonstrator Schedule Object

As a novice HCI designer, a particular problem that I experienced was trying to make a connection between the ***Task Model*** and the ***Object Model*** in the demonstrator documentation. There appeared to be very little formal link between the two models, and time was spent trying to find one. A document which was linked more closely with the ***Object Model*** was the Feasibility report.

A traditional approach to developing object models is to analyse a textual description and highlight key nouns. The nouns are then reviewed and the key ones filtered out as objects. The feasibility document contained a description of the intended functionality of the Project 1 software, so it was be used to derive key nouns from which, the main objects were sifted out and compared with the existing ***Object Model***. This was duly done but the Demonstrator's ***Object Model*** soon became little more than a starting point (almost a red herring). Organising the basic conceptual objects of the software into a structural overview was surprisingly difficult. Reference material on OOD was geared towards producing code for given simple examples rather than designing a user interface (UI). However, I believed that I was specifying the UI through first getting the structure of the software right and not specifically concentrating on just the visual design. This view was contested by software designers, programmers and the project manager during the early phases of the development as the HCI designer's role was being defined. Using the basic conceptual objects I constructed an Entity-Relationship Diagram (ERD) (which is a database design technique that I was conversant with) to show at a high level, how the objects would fit together. Interestingly, one of the software designers, SD/P_2, also new to OO techniques, produced a Class Diagram (OOD representation) of the Project 1 software several months later, which was identical to my ERD, in all but notational detail and apparently without direct reference to my diagram. The ERD was a useful starting point for a more detailed object analysis and modelling. The final ***Object Model*** consisted of 4 sections, *Object*, *Action*, *Prerequisites* and *Results*, as can be seen in the work sample shown in figure A.4.



Object An'ysis

Carl Myhill 16th November 19[illegible]

OBJECT	ACTION	PRE-REQUISITE	RESULT
:spec sheets	[select]	select valve assign units standard	Display spec sheet. This may be blank if kb has not yet run.
	edit	spec sheet selected	compare to previous revision - red ▵ denote changes Notes section updated with change reasons
	print	select valve assign units standard	printout, showing various triangles (?) and notes
	create kb	project selected	User names the kb Create and Display a template listing all fields - showing all areas as blank User may specify values for a field by selecting a field, viewing a list of possible values for the field selecting from this list
knowledge base	run	select valve (sufficient) process data exists valve performance kb assigned	If spec sheet is already completed, warning should be given that data will be lost and a cancel option offered - if OK overwrite previous spec sheet. Indication must be given to the user that the kb has run and the outcome of that run - 'successful specification', 'unable to specify', etc. Display Spec Sheet
	query for trace	select valve run	Display trace and explanation

Figure A.4 Work Sample of an Object Model Segment Centred on the Project 1 Specification Sheet Object

This style of ***Object Model*** was considered poor as a standalone document to describe the conceptual model of the proposed software. However, it proved useful as a working document during discussions about proposed functionality with the technical manager. Unfortunately, the representation did not lend itself to quick interpretation and it appeared to take around 30 minutes of discussion and analysis to re-comprehend the model before effective discussion about conceptual functionality could take place. This was a significant amount of time, particularly as the object model underwent around four main iterations. However, in spite of the uncommunicative nature of the representation, the ***Object Model*** was the first working document I produced which was directly handed over as an input to the software designers, in this case SD/P_1. From this document SD/P_1 went on to produce more detailed documentation, notably Data Flow Diagrams (DFDs) which showed proposed data flows around the software being designed.

DFDs initially produced by SD/P_1 proved to be a difficult representation to apply to the internal design of the Project 1 software. The primary reason for this is that Windows is an event driven environment and DFDs were designed for data driven environments. The main problem with the approach is that a DFD should strictly only show data flows and processes and should not show control. This means that the representation is not designed to show button presses, menu selections or other Windows events. This problem was dealt with by adding control flows to the diagram, denoted, e.g. [OK button pressed], this complicated the diagram and was perhaps an inappropriate extension to the representation. A second problem for the software designers was that in Windows, the user has many degrees of freedom of actions that can be performed at any one time, e.g. it would be unusual in Windows if the user could not choose to do over 50 different functions at any point, which clearly creates a design problem. During this design phase the software designers looked around for alternative notations, especially OOD notations but they did not prove to be easily accessible techniques even to software designers like SD/P_1, with over ten years experience in design. Given the time constraints of the project it was felt that designers should use the techniques and representations that they knew well and adapt them for use with an event driven Windows environment. Fortunately, having a software background I was conversant with DFD representations and was able to contribute to and review designs represented in this way. At this stage, SD/P_1 in particular felt that DFDs and structural issues were nothing to do with an HCI designer.

December 1992

Following the production of an ***Object Model*** I was charged with producing a ***System Map*** which was supposed to be a single page diagram giving an overview of what the software would do. The ***System Map*** in the original documentation was fairly broad and mapped out what a comprehensive control valve specification system should comprise, e.g. including vendor catalogues, names and addresses and other periphery. The Project 1 software was to be a more cut down system than this, and was primarily intended to show that the KBS core of the program could produce sensible suggestions about specifying valves for given process conditions. Therefore, a new ***System Map*** was needed to show this cut down system. I ultimately concluded that the best system map I could produce was the ERD that I had previously used as a stepping stone to producing the ***Object Model***.

From the initial user interface design plan (shown in figure A.1) nobody realised what the ***Screen Mock-ups*** stage of the process involved. From my perspective, I had by now gained an understanding of the user's tasks, the intended functioning of the KBS, and some idea of the basic conceptual building blocks of the Project 1 software. Essentially what followed was a blank sheet of paper where a visual design needed to be. The major piece of advice I was given was from the technical manager (also acting as elected 'pseudo client/user'), who said that the Project 1 software should have a similar "look and feel" to the Demonstrator software. I felt that this was wrong as the end-user of the Demonstrator was the company salesman, whereas the end-user of the software was to be an actual engineer specifying valves for real. Even with this advice the blank sheet of paper remained, in fact, the problem was possibly worsened by technical manager's advice which was effectively a further constraint.

The approach that I ultimately adopted to get started was to re-consider each object and its associated actions and try to translate these into Windows artifacts. At first, preliminary block diagrams were sketched to show visually what the key elements of the Project 1 software would be. These diagrams contained little more than boxes representing windows, titles and samples of the type of information that the window would contain (where this added clarity).

The process of producing a visual user interface design for the software proved to be trying. At the outset, I sketched block diagrams of aspects of the software autonomously and then reviewed these ideas with the technical manager. The next step was to produce a high level visual walk through of the Project 1 software by developing a storyboard sequence of a major route through the software. This was a difficult task to have confidence in, as at this level of abstraction it was clear that there were many gaps in the current conceptual model. However, I first tried to represent the storyboard by arranging all of the sketches on an A2 sheet of paper. There followed a design review by the project team (by now consisting of the project manager, technical manager (acting as 'pseudo client/user'), three software designer/programmers (SD/P_1, SD/P_2, SD/P_3), a knowledge engineer and three programmers (P_1, P_2, P_3)) which was disjointed, unstructured and unsystematic. The A2 sheet representation let the team see the whole picture all at once, allowing people to continually pick up on points that were not relevant to the flow of the walkthrough. The result was that inappropriate details were discussed instead of the overall structure of the visual interface. To counteract this, I decided to adopt a more 'advent calender' or 'origami' style approach to the walk-though for the next design review. This involved focusing the audience's attention on one screen sketch at a time and showing interaction by arranging paper in various ways, e.g. folding down stuck on menus, or placing cut out dialogs on the main sketch. This approach proved far more effective than the 'all at once' storyboard but was far from perfect. Documenting the interaction ultimately required a significant amount of additional work as the main specification for the interactivity was only available from a pile of cut out pieces of paper. Recording the comments made by project team reviewers, whilst at the same time trying to fold down the next menu, or arranging a set of 3 cut out dialogs in a certain way was difficult. As was the critical task of controlling the versions of the multitude of pieces of paper which comprised the current design intent.

January 1993

Although the initial user interface development plan (shown in figure A.1) suggested that detailed screen design was part of ***screen mock-ups***, it began to evolve into a major activity in it's own right. At this time, the plan being followed to develop the user interface was forgotten and had been replaced by a concerted effort to finalise the ***detailed design*** of the Project 1 software.

Detailed design of the software was based on the final version of the storyboard walkthrough and incorporated comments that had stemmed from it. There was no definite start to this phase of the development. I felt the need to add more detail to the sketches used in the walkthrough in order to better illustrate concepts, so began to add more detail. When we realised that the ***detailed design*** activity had started, perceptions and expectations of what it involved were initially quite different. Early on in the process the project manager was very keen to know when the ***detailed designs*** would be finished, so that the programmers could get on with their work and my involvement would be over. The thought of my involvement in the project being over only half way through the development confused me somewhat as I could still see many tasks that needed to be performed by someone responsible for the interests of the users. At this stage, the project manager and the programmers still saw the HCI designer as someone who arranges information on a display - the information having been dictated by programming needs following a functional specification. In my view, I was defining exactly what information would appear on screen, when it would appear and how it would be manipulated based on the user's tasks and requirements and the computer would just have to deal with it. Ultimately we learned to compromise but each party usually felt wronged by compromising their ideals, for me this was the ideal interaction, for programmers the ideal was elegant code. At least other team members now saw me sketching, a task which they saw as my main contribution, so they were a bit happier now that I was not meddling with the structure of the software (in their eyes). However, it soon became apparent to me that what I was actually doing during the ***detailed design*** phase was specifying in detail how the Project 1 software should work. This was not at the level of deciding on icons for warning messages (although this was part of my role too) but rather deciding on functional mechanisms that, for example, would enable the user to open a new project or select a piece of equipment.

The ***detailed design*** activity itself was very demanding for several reasons. Firstly, the activity required considerable creativity to design the software within the constraints of the Windows environment and with the limited Windows and C++ skills of the development team. Secondly, I had very limited sketching ability, so representing complex concepts with pencil and paper was infeasible. Thirdly, the ***detailed design*** activity became a project bottleneck as the programmers could not begin to code until the design was completed. This led to some tension between the project manager and myself, as he regularly asked for me to complete the design for at least some parts of the software, so that the programmers could at least work on these areas. I resisted this as I treated all of the designs as provisional until I could see how the whole software would hang together. Ultimately, the design was largely passed to programmers in sections, and worse still, the programmers began working on the internal structure of the software based on a rudimentary understanding of the preliminary walk-through and with little consultation with me about the UI. Part of the reason for this bottleneck and the resulting tension was the fact that producing the detailed screen designs took a long time. My inability to sketch ***detailed design*** concepts effectively led me to using rudimentary computer tools to paint screens. The approach adopted was to screen capture a Windows artifact (e.g. a dialog box) from another application, copy this into Microsoft Paintbrush (a primitive bitmap editing tool) and edit the bitmap. This approach, which provided a design which looked like a credible Windows artifact, seemed to convey more to the team, especially programmers, than a pencil sketch had done previously.

February 1993

The availability of Microsoft Visual Basic (VB) during the later half of the ***detailed design*** phase allowed some prototyping to be tried out. Conveying the results of the KBS valve specification was perhaps the most difficult aspect of the design. As a new HCI designer this posed a particular problem as I could find nothing remotely similar to look at, within any other Windows application that was available, so I had to innovate. Visual Basic provided a fast method of mocking up a screen but it was also sufficiently powerful to show interaction of Windows artifacts in a realistic way (Visual Basic is not just a prototyping tool, it can also be used to generate powerful Windows applications relatively quickly) . VB proved immensely useful as a design aid, in particular to ease the burden of visualising (and conveying visualisation to others) a complex sequence of activities by actually animating them and also allowing design experimentation. Prototyping was beginning to look like the ultimate way of communicating within the design team and would solve many of the communication problems that had previously arisen. However, there soon became apparent some dangers of reliance on the prototyping method.

The nature of the prototyping tool, Visual Basic (VB) and that of the development language, Borland C++ (BC++) were very different. This meant that some functionality which could be easily mocked up in VB may take a considerable amount of time to replicate in BC++ and vice versa. On occasion this led to tension with programmers as my designs could quickly become technically ambitious on the one hand, or technically limited on the other, according to my understanding of Windows and BC++. Eventually P_1, one of the more experimental programmers (i.e. a hacker) proved a useful resource for me, as I could chat to him about the feasibility of achieving a certain effect in BC++. Whilst this person had a fairly unstructured approach to work and many unrealistic attitudes towards timescales, it was possible to get a feeling for how hard something would be from him. Other programmers on the team, e.g. P_2, would often say that something just could not be done, rather than admitting that they did not know how to do it. On one occasion I was told by SD/P_2 that something would take three weeks, even though he admitted that he had not yet investigated how it would be achieved.

Programmers also appeared to have a great reluctance to look at the prototype at all, which I ultimately found very frustrating. Programmers never asked to look at the prototype, they would usually ask a specific question about how something should be achieved which I would answer using the prototype to demonstrate.

Another problem with prototyping was that the approach did not effectively capture design rationale. During the development phase it became clear that by deviating from the design intent in certain small ways, the job of the BC++ programmer could be greatly simplified. However, what actually happened was that the programmers would read the specification document and would recollect some basic intent specified by the prototype and would use this as a sufficient mental model of what the software should look like. They would then use whichever artifacts were convenient in BC++ to achieve functionally equivalent results to the prototype. Whilst this was an acceptable approach when the programmer consulted with the HCI designer and mutual agreement was reached so design rationale still held, when the programmers made autonomous decisions about slightly different implementations, they would often not be aware that the changes had compromised design rationale.

A further problem was that when presenting a prototype as a structured walk-though to developers, they would often appear to focus on information specific to their direct task. For example, programmers appeared to be more interested in what the data was on a form and where it had come from, rather than the mechanisms which controlled the manipulation of the form. Even where the data were simple, and the form manipulation complex, it was difficult to convey the importance and complexity of the form manipulation over the data.

An example of this problem was dubbed, ‘the Mexican wave’. SD/P_2 had to construct part of the complex results module of the software. The module was centred around a grid, which was to scroll left and right to show more information. Being new to Windows programming and BC++ the programmer opted to use tried and tested conventional techniques to achieve this functionality, using the scroll bars as mere buttons to run a procedure. When the scroll bar was pressed, the procedure would start at the top left of the screen and copy each cell into its neighbouring column. Whilst the end result was functionally correct, it was unfortunate that the copying of each cell to its neighbour was visible to the user in the form of a delayed ripple across the screen (hence the name ‘Mexican wave’). Having focused on the data and not on the mechanism, the programmer had achieved the ‘correct’ functionality (strictly, this would be sufficient to meet the Specification) but the visual appearance of the interaction was clearly unacceptable. Unfortunately, it was not always clear to everybody that an implementation which was strictly functionality correct according to the specification was not acceptable in the interaction sense.

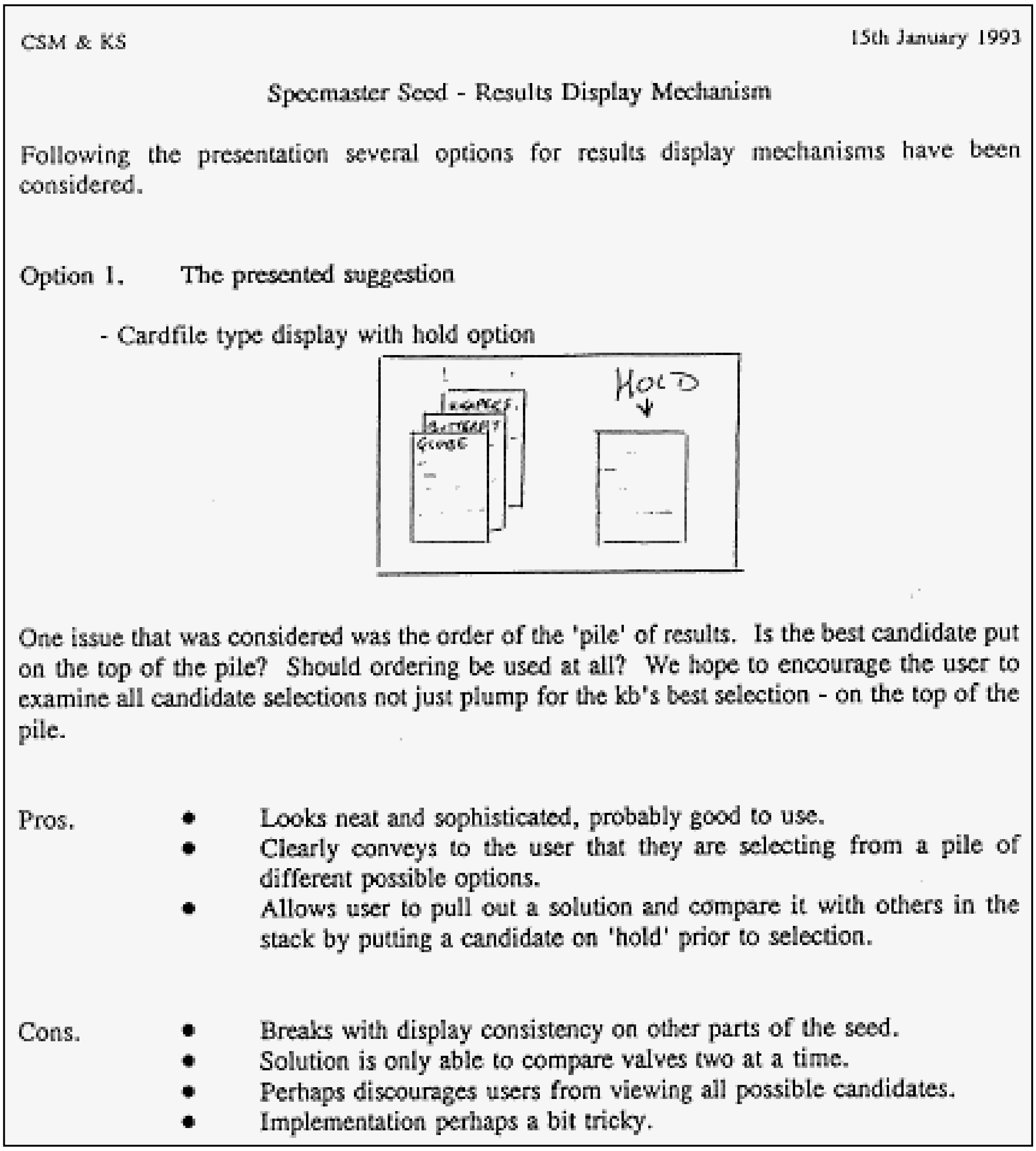
A final problem with the prototyping approach is that it is possible to spend large amounts of time perfecting a design. At times I isolated myself from the team by becoming too engrossed or too perfectionist with the prototype.

March 1993

Ultimately, the process of designing elements of the software became refined. Whether represented by sketch, screen bitmaps or prototype, the process involved in designing a mechanism in the software, such as the results module, would usually begin with a team brainstorm. The brainstorm was intended to satisfy the philosophy introduced by the technical manager which aimed to always produce at least three options to any design problem. Following the brainstorm the HCI designer would mock-up the various solution ideas and a meeting would be held to review these and select the best one. I soon discovered that the best method of documenting possible solutions involved writing a simple one paragraph description with a visual representation and a pro/con analysis of the option, as can be seen in figure A.5. This approach was geared towards presenting and explaining each option concisely so that a decision could be reached, rather than writing up the options for team members to read for themselves.

It was often difficult to agree on the most appropriate mechanism to meet the requirements, possibly because the project requirements were not clear and were often interpreted differently by each team member.

Requirements from the perspective of each of the disciplines represented in the team were also different. As an HCI designer I considered the user requirements foremost. Software designers were more interested in a solid structure to the software which could be easily constructed and maintained. Similarly, programmers were motivated by their need to actually construct what was being designed and in an elegant way. The project manager, motivated to produce the software to Time, Budget and Quality (TBQ) was therefore keen to ensure that each designed mechanism would be viable to construct in the time available and be of sufficient quality. The project manager always appeared to want to minimise the functionality and work effort required to build the software as will become apparent later.



CSM & KS 15th January 1993

Specmaster Seed - Results Display Mechanism

Following the presentation several options for results display mechanisms have been considered.

Option 1. The presented suggestion

- Cardfile type display with hold option

One issue that was considered was the order of the 'pile' of results. Is the best candidate put on the top of the pile? Should ordering be used at all? We hope to encourage the user to examine all candidate selections not just plump for the kb's best selection - on the top of the pile.

Pros.
- Looks neat and sophisticated, probably good to use.
- Clearly conveys to the user that they are selecting from a pile of different possible options.
- Allows user to pull out a solution and compare it with others in the stack by putting a candidate on 'hold' prior to selection.

Cons.
- Breaks with display consistency on other parts of the seed.
- Solution is only able to compare valves two at a time.
- Perhaps discourages users from viewing all possible candidates.
- Implementation perhaps a bit tricky.

Figure A.5 Work Sample Showing the Documentary Representations Used to Put Forward Design Ideas

April 1993

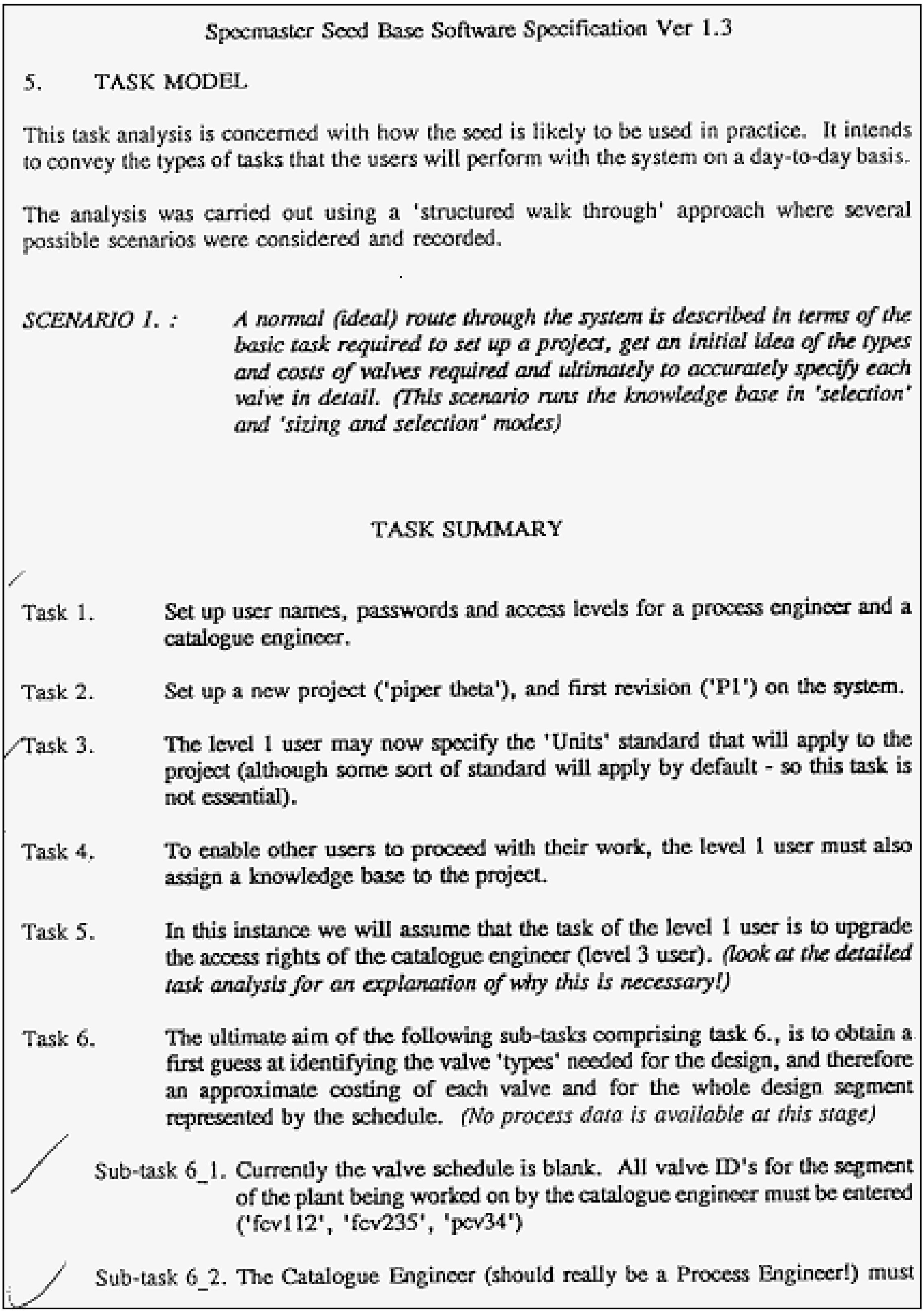
The first major project milestone was the production of the Software Specification. So significant was achieving this milestone and getting the Specification signed-off by management, that the project manager bought a cake and the team enjoyed a company celebration and speech. Whilst this may have been a team motivation exercise, it did bring home the importance of Project 1 to the company.

The structure of the Specification document was taken from guidelines previously used by Amtech and was fairly dated. Each section of the Specification was produced by the relevant disciplinary area. As the HCI designer I was responsible for the screen sketches and the task models, the software designers were responsible for the software requirements and functional descriptions (which referred to the screen sketches) and data descriptions. The knowledge engineers produced details of the workings of the KBS core of the software. Every section in the document was written in each individual's own preferred font and style. The editing of the Specification concentrated on checking the correctness of information in the document rather than presentation and readability.

Producing the Specification was seen by most team members as a chore, particularly as it meant committing themselves to a design that they were as yet unconvinced about. From an HCI designer's perspective, this meant finalising screen designs at a high level of detail, knowledge engineers had to baseline their thinking on the knowledge model and software designers had to complete a suitable software structure. It was understood that the Specification could not represent a final reflection of the Project 1 software at such an early stage in the development and it was therefore open to amendment. However, the Specification was signed-off at the highest level inside the company and was used by the project manager as a kind of contract to what had been agreed we would produce. In later discussions, when further shaping the software, adding details or introducing new ideas, the project manager, ever conscious of time and budget, would often put forward a minimalist attitude towards changes, usually accompanied by the phrase, "what are we committed to produce?"

During the exercise of writing the Specification it became apparent that team members were still unclear how the Project 1 software would look and behave. Even though a partial prototype existed and by now several iterations of each screen design had been achieved, it was still difficult to visualise the proposed software in use. It was felt that the task models should have solved this problem but in their existing format did not provide what was required. As the HCI designer I was charged with looking at how the current conceptual design would facilitate the engineer's tasks. The approach taken to address this problem, was to produce a model of how the engineer's task would be structured if the software were in place, this was called a *Future Task Model* (FTM). I devised this representation *ad hoc* to solve a particular problem. Whilst possibly not a unique idea it highlights the fact that, in the role of HCI designer a person needs to be flexible, understand the needs of the team and come up with solutions some how. Practical texts, with relevant, applicable and commercially viable techniques or approaches were not available.

The FTM comprised a basic scenario (consistent with the engineer's tasks as we saw them), a summary of the main tasks necessary to achieve the scenario, followed by a detailed breakdown explaining how the engineer would proceed through the main tasks using the proposed software (at a level of detail, specifying for example, which buttons would be pressed). These features can be seen in figures A.6 and A.7.



Specmaster Seed Base Software Specification Ver 1.3

5. TASK MODEL

This task analysis is concerned with how the seed is likely to be used in practice. It intends to convey the types of tasks that the users will perform with the system on a day-to-day basis.

The analysis was carried out using a 'structured walk through' approach where several possible scenarios were considered and recorded.

SCENARIO 1. : *A normal (ideal) route through the system is described in terms of the basic task required to set up a project, get an initial idea of the types and costs of valves required and ultimately to accurately specify each valve in detail. (This scenario runs the knowledge base in 'selection' and 'sizing and selection' modes)*

TASK SUMMARY

Task 1. Set up user names, passwords and access levels for a process engineer and a catalogue engineer.

Task 2. Set up a new project ('piper theta'), and first revision ('P1') on the system.

Task 3. The level 1 user may now specify the 'Units' standard that will apply to the project (although some sort of standard will apply by default - so this task is not essential).

Task 4. To enable other users to proceed with their work, the level 1 user must also assign a knowledge base to the project.

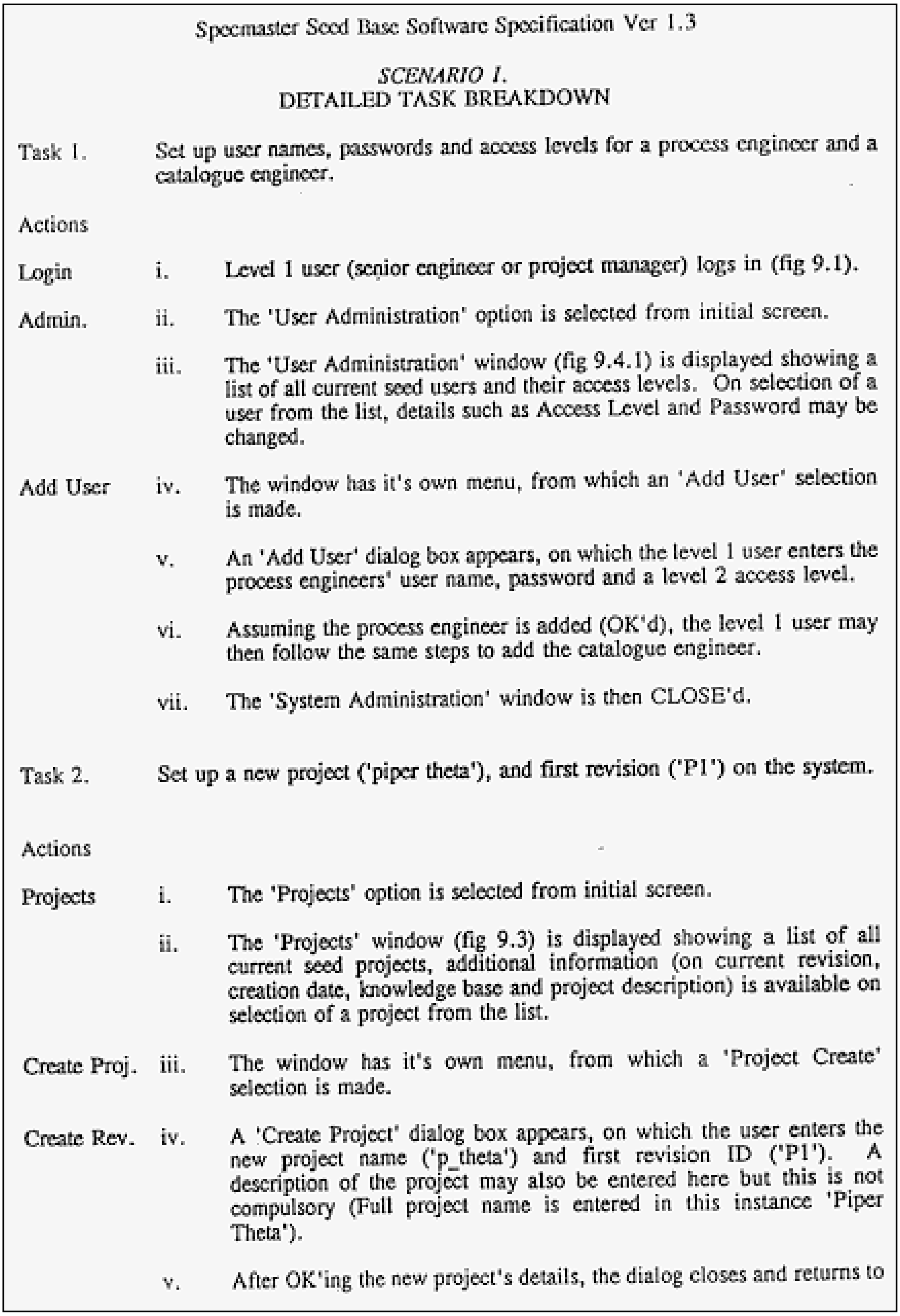
Task 5. In this instance we will assume that the task of the level 1 user is to upgrade the access rights of the catalogue engineer (level 3 user). *(look at the detailed task analysis for an explanation of why this is necessary!)*

Task 6. The ultimate aim of the following sub-tasks comprising task 6., is to obtain a first guess at identifying the valve 'types' needed for the design, and therefore an approximate costing of each valve and for the whole design segment represented by the schedule. *(No process data is available at this stage)*

Sub-task 6_1. Currently the valve schedule is blank. All valve ID's for the segment of the plant being worked on by the catalogue engineer must be entered ('fcv112', 'fcv235', 'pcv34')

Sub-task 6_2. The Catalogue Engineer (should really be a Process Engineer!) must

Figure A.6 Future Task Model Showing Scenario Description and Task Summary, Taken From the Software Specification



Specmaster Seed Base Software Specification Ver 1.3

SCENARIO 1.

DETAILED TASK BREAKDOWN

Task 1. Set up user names, passwords and access levels for a process engineer and a catalogue engineer.

Actions

Login i. Level 1 user (senior engineer or project manager) logs in (fig 9.1).

Admin. ii. The 'User Administration' option is selected from initial screen.

iii. The 'User Administration' window (fig 9.4.1) is displayed showing a list of all current seed users and their access levels. On selection of a user from the list, details such as Access Level and Password may be changed.

Add User iv. The window has it's own menu, from which an 'Add User' selection is made.

v. An 'Add User' dialog box appears, on which the level 1 user enters the process engineers' user name, password and a level 2 access level.

vi. Assuming the process engineer is added (OK'd), the level 1 user may then follow the same steps to add the catalogue engineer.

vii. The 'System Administration' window is then CLOSE'd.

Task 2. Set up a new project ('piper theta'), and first revision ('P1') on the system.

Actions

Projects i. The 'Projects' option is selected from initial screen.

ii. The 'Projects' window (fig 9.3) is displayed showing a list of all current seed projects, additional information (on current revision, creation date, knowledge base and project description) is available on selection of a project from the list.

Create Proj. iii. The window has it's own menu, from which a 'Project Create' selection is made.

Create Rev. iv. A 'Create Project' dialog box appears, on which the user enters the new project name ('p_theta') and first revision ID ('P1'). A description of the project may also be entered here but this is not compulsory (Full project name is entered in this instance 'Piper Theta').

v. After OK'ing the new project's details, the dialog closes and returns to

Figure A.7 Future Task Model Showing Detailed Task Breakdowns, Taken From the Software Specification

The FTM was 13 pages long and produced in one afternoon based on my mental model of the proposed software and my understanding of the engineer's tasks. The FTM was much acclaimed and several team members, in particular the project manager claimed to have gained a greater understanding of how the software would work from this representation. Of particular use were the scenarios and task summaries which showed how the engineer's tasks would be facilitated by the project 1 software. The detailed breakdown of precisely which buttons would be pressed to achieve the tasks in the summaries was generally thought to be too much detail. Despite the apparent effectiveness of the FTM at showing how the software would be used in practice, there was resistance to putting it in the Software Specification. It was considered unconventional to put detailed task models in a Specification but ultimately seen to be of value by senior members of the team, in particular the project manager.

Following the production of the specification document, the project team grew to around 10 people. Weekly project progress meetings became a lengthy and often frustrating, so the project manager decided to split the team into 3 disciplinary groups for the meetings. Whilst this made the meetings shorter and more manageable, as the HCI designer I felt that I lost touch with the rest of the team considerably at this time. In particular, I became very distanced from what was going on in the KBS core of the software which was no small concern as this element of the team had closer contact with the industry (and indirectly with the users) than anyone else.

May 1993

Following the production of the specification document, the emphasis changed from design to construction of the software. At this stage, many of the visual designs were still vague and would require more thought, but the major structure of the Project 1 software was becoming clear. As ***detailed design***s became finalised for a particular area, that part of the software would be handed over for implementation. I often resisted handing over designs until the last possible moment as I was conscious of the fact that other aspects of the software had not yet been finalised and would probably have knock-on effects to all other areas. Consequently, this resistance to letting go of a finalised design, or of being perfectionist on other aspects, caused considerable delays on the project and the HCI ***detailed design*** activity became an obvious bottleneck. It was clear to me that I was causing the bottleneck and was on the critical path of the project, I was the only person involved in designing the visual and interaction detail of the software but there were around five programmers waiting to pick up my designs and get to work. This became a high pressure period on the project for me, which involved a considerable number of late nights and extra effort. The problem ultimately worsened as programmers began to work on or complete various aspects of the design, because this meant that I had to answer a stream of questions about the design, monitor what was being implemented and continue the ***detailed design*** of other areas. It felt like feeding and looking after half of the project team.

Although the process of implementing a ***detailed design*** element is only a small part of the overall project activity at any given time, considering a single element in isolation will facilitate illustration of the process. A completed design would be passed to a programmer and depending on who that was, several things may happen. Some programmers would use the given information, make any further assumptions themselves and implement the design, without referring to anybody else. Others would look at the design and see it as infeasible given the commercial constraints and may suggest an alternative to me, which would start a negotiation process involving new options and compromises. Others would ensure that the implementation fulfilled its functional specification, i.e. it provided the right data at the right time, but saw any further work on the visual appearance of interaction as unnecessarily finicky. Frequently, various aspects of a design proved infeasible after the implementation had commenced and this necessitated some re-design effort. As making changes to the specification and the design became somewhat of a contentious issue, I labelled these changes, 'implementation driven re-design'.

Changes in requirements from the Specification document or from the currently accepted designs proved a considerable disturbance to the development process, both in terms of the extra work generated and the increased tension and frustration in the team. Some of the requirements changes were generated from contact with potential clients, either through the salesman or through knowledge engineers who were more closely linked to the project. Knowledge of the client domain was still in a formative stage during the project and the Process industry was being hit hard by the recession, so the requirements of the clients themselves were also changing. Some changes were driven by the fact that the design could not be implemented in the intended way for a technical reason. Other perceived changes were generated by the review process when assumptions had been made and design intent not followed. Some changes were deemed necessary following the implementation of certain aspects of the design which when fully working did not feel right. Other changes still came from within the team as people gradually got a better grasp of the problem or suddenly thought up a better way of doing something. It appeared that the programmers always looked upon changes as a negative thing.

June 1993

It should be noted that with the early August deadline approaching, the team, in particular, the technical core, became furiously busy finalising the implementation.

As the implementation phase got well under way, my role on the project changed from designing to reviewing the UI. In fact, the role became very similar to that of the project manager, in that I was keeping close tabs on the implementation effort, ensuring consistency and quality, as well as contributing to decisions on implementation priorities. In some ways I became a kind of technical liaison between the project manager and the technical core of the team enabling me to exert user centred influence on decisions made.

Throughout the project, the process of reviewing aspects of work was not clearly defined or well structured. This was particularly the case when it came to reviewing the UI of implemented elements of the software. I attempted to monitor progress that the technical people were making in an informal way in the hope that I catch any misunderstandings early and they could rectified easily. However, several members of the technical core of the team were not open to this approach and would dismiss any guidance or corrections that I made informally, apparently thinking that I was telling them their jobs. Informal monitoring was also hampered by the fact that programmers' code seemed to be in a continual state of development and that at any particular moment in time the code was not in any fit state to run (analogous to trying to test drive a car with no wheels!). Thus, implementation often proceeded unchecked and it was left to the formal review process to pick up on any problems. The formal UI reviews were put off by programmers until the last minute, I believe partly because they knew that the later a mistake was discovered, the less likely it would be that they would have to make changes, as the deadline was approaching. UI reviews would often very quickly get programmers into a defensive stance, who would then begin to attribute blame for any misunderstandings. I was often criticised for being too finicky about the UI, and any changes I suggested were often resisted. Fortunately, by this time, the project manager realised the benefits of HCI and my new role on the team. I think in particular, he knew that my concern for the UI was closely aligned with his concerns for the quality of the software. It was because the project manager decided to champion the HCI cause and support my user perspective that I was able to get any changes made at all. Ultimately, as the project deadline got nearer, myself and the project manager would often discuss and prioritise remaining work, particularly with respect to bugs and enhancement requests. The project manager's influence enabled HCI bugs to be viewed with equal severity as technical bugs, which affected the team in no small way!

When bugs were discovered in the implementation, I was surprised at how the programmers took them personally. On discovery of a bug, some programmers were keen to attribute blame for it, seemingly to clear their own name. In some ways this seemed to emphasise the individualist rather than team approach to the development.

Of the 250 bugs that occurred during the Project 1 software development, the severity and number of bugs could have been reduced had there been fewer misunderstanding and misinterpretations, greater team cohesiveness and more relevant skills and experience than was evident in the Project 1 software team. I believe that better team skills and communication and a less individualistic approach could have dealt with the bugs more effectively. It is worth bearing in mind that number and severity of the bugs in the Project 1 software would not be considered unusual in the software business - nor would the contributory factors and solutions recommended in this paragraph.

My popularity on the project suffered during the latter stages of the project as a result of my inexperience as an HCI designer. On reviewing some of the implemented elements of the code, I was able to see some of the designs animated in their interaction for the first time. Seeing the implemented UI I was able to criticise my design more effectively than I had before, which resulted in me requesting changes to the implementation. Needless to say, this did not go down too well.

July 1993

During July, the technical core of the team began to integrate individual's code elements into a single software version, known as a baseline. The software gradually progressed through several baseline versions each integrating newly completed code elements, up until the deadline. Previous baseline versions were stored away, with the latest one being a fall-back position should the current integration fail. From the first integrated baseline, testing began.

Formal testing was carried out by the technical core of the team and took on several forms including ensuring that the KBS was providing correct results, functional testing of modules and functional testing of implemented baseline. Informal testing was carried out by myself (in order to pick up on any UI problems), the project manager and anybody else who happened to have some spare time. Informal testing was relatively haphazard, feedback from which resulted from people spending time playing with software and trying to break it. Informal HCI testing at this level consisted of ensuring that design intent had been followed, rather than any user evaluation. Task models, in particular the FTM were walked through to ensure that predicted user tasks could be adequately facilitated by the software.

Following testing, more bugs would usually be found resulting in more work for the technical core of the team. Time was running short by now and bugs were carefully prioritised and the work of the technical core of the team carefully monitored to ensure that time was not wasted working on low priority problems.

August 1993

The Project 1 software was delivered on time in early August following considerable extra efforts and long hours by the team in the closing two months. By this stage, some low priority bugs remained in the software and some further enhancements and new requirements were outstanding.

A baseline version of the software that was sufficiently robust for user testing did not arrive until a few days before the deadline. An evaluation of the Project 1 software did not take place until after it had been delivered. However, at the outset of the project it was clear that the software was itself part of the formative stages in the development of the ultimate product.

The evaluation was carried out with the technical director of BHRG who had not been directly involved in the software development but had thought up the Project 1 idea in the first place and had good general understanding of engineering and the software domain. The pseudo-user was not typical of the proposed end users of the Project 1 software but his input turned out to be of value nonetheless. The evaluation was task-based and involved observation of the user working through predefined sample scenarios where a Control Valve selection was necessary. From the observation, some elements of the UI design clearly did not inform the user where to proceed next. It was a shock to me, having considered the UI design of the software for many months, to discover basic errors in the design. After such considerable implementation effort, it was now infeasible to significantly redesign the UI but minor ‘alterations’ were made to overcome some of the problems discovered. As well as observation, a self-completion evaluation checklist was also handed to the user. Commercial constraints on the time of the Technical Director did not provide him with the opportunity to complete the 13 page questionnaire.

Development of the Project 1 software continued until November with a much reduced team that was fixing outstanding bugs and adding new functionality to the software, for example, a KBS editor was added for easy maintenance of the KBS.

Epilogue

From the narrative, it is apparent that the ***Windows Design Guidelines*** and ***Design Rationale*** activities introduced in figure A.1 and table A.1 did not feature strongly in the development process.

It was my responsibility to be aware of Windows design principles during the project in order to ensure compatibility with other Windows software. A brief two page style guide was produced for programmers to use so that their initial implementations would adopt the correct font, button sizes and positions, etc. This style guide apparently quickly became buried amidst the mass of project documentation on the desks of the team members, and so was not used extensively.

Design rationale as documented for the Project 1 Demonstrator consisted primarily of a description of the major elements of the proposed software. ***Design rationale*** for the Project 1 software itself was not effectively captured. At times during the project when changes were required to the UI, I kept a log book documenting the rationale for the changes and the new design. However, this log book was only a personal reference and was not a team resource.

Appendix B Detailed Structured Analysis of the Project 1 Chronology and Other Data

1.1 Introduction

From participation-observation of Project 1, dominant influences effecting the software development were apparent. This section discusses the influences which appear to have had the most significant effect on the software project. Influences identified include characteristics of the project itself, the structure of the software organisation, the development team, representations and techniques adopted, the software specification, the development lifecycle, characteristics of individuals in the development team and the effect of changing requirements.

1.2 Overview of Project 1

The Project 1 software was to be a speculative product development which was intended to try to convince an industry typically cynical about knowledge-based software, that intelligent computer support for equipment specification could be useful. The philosophy of the Project 1 venture was to produce a 'Seed' system which would be sold as an ongoing development, allowing potential customers to work with SFK to configure the software with their own knowledge, standards and guidelines. This philosophy evolved from the company's need for more detailed and specific domain knowledge required to build complex KBS modules to aid equipment specification. The control valve module was selected for the Seed software as it was an area where the software company already had some experience and further domain knowledge was accessible.

The concept for the Project 1 software had been around in BHRG for 20 years but a year before the Seed development began an in-depth Scope study was produced. The Scope study set the scene for the software development and recorded the software company's understanding of the domain and the potential context of the software.

Following the Scope study, with input from management, sales and technical staff, demonstration software was constructed. The demonstration software was used by sales staff to sell the concept of the Project 1 software to the industry and pave the way for potential Seed contracts. Constructing the demonstrator software had other motivations, as it gave the software company the opportunity to build their first Microsoft Windows application and their first C++ application. Thus, the demonstrator allowed the software company to gain new skills in the design and development of Windows applications using Object-Oriented (OO) programming techniques.

The Seed software project started on 23rd August 1992 and was signified by the production of a Seed feasibility document which was the main reference for the early months of the software development. This document, along with the original Scope study and the demonstrator software formed the documentary starting point of the project. This core information was supplemented by Visit Reports which documented visits to control valve experts at ICI and other relevant industrial contacts.

There were several key characteristics of Project 1 that should be considered. The Seed was innovative and complex software with a KBS core and was a speculative venture, so there was no actual client and no direct access to end-users.

Although the structure of the team was to change throughout the development process, several features of the team's skills at the outset of the project are relevant. The majority of the programmers on the team had very little experience of producing software under a Microsoft Windows environment, were new to the development language (Borland C++) and Object-Oriented Design (OOD) and programming techniques. In software terms, it is a very significant leap for a programmer to switch from structured programming in an environment where the code requests the input, to an OO event driven environment where the input requests the code. The HCI designer was new to the HCI design role and had just joined the software company. The structure of the team and responsibilities changed the software company's typical development team with the advent of this new role.

1.3 Project Characteristics

There are a number of characteristics of the Project 1 Seed software which appeared to have a direct influence on the software venture, these are summarised in table B.1 and explained in more detail the following sections.

Table B.1 Observed Project Characteristics and their influences and effects on Project 1

Project Characteristics	**Project 1 Indicators**	**Apparent Effects**
Project goals and objectives	• No client • Speculative and innovative Seed venture • Disagreement between technical and sales areas • Domain knowledge too low to specify requirements	• Team member's understanding of objectives not aligned • Requirements changed as exposure to the domain improved
Skills, experience and competence of team members	• Technical staff had very little knowledge of: • Microsoft Windows • Borland C++ • Object-Oriented Design • HCI designer was inexperienced • Competence of team members was variable	• Unknowns in software design and implementation approaches • Application style required for development environment not well understood • Unknowns in HCI design activity as the designer inexperience meant that he had no heuristics on which to base judgements, e.g. estimating work effort
Team experience of working together	• HCI designer was new to the team • Knowledge engineers new to the team • Software designer/ programmers and project manager had previously worked together	• Cohesion of team low. • Roles and responsibilities of individuals within the team evolved during the project • User-centred rather than technology-driven approach to project.
Understanding of the domain	• First-hand domain knowledge mainly held by knowledge engineers and sales staff. • Second hand domain knowledge sought by HCI designer, project manager and KBS programmer. • Domain knowledge NOT sought by software designers or programmers.	• General understanding of requirements held by team members was not consistent or cohesive. • Individual's conceptualisation of what the software would look like and what it would do were often different. • Some individuals appeared not to try to conceptualise the whole system but would prefer to concentrate on conceptualising solely the element that they were working on.

Table B.1'continued Observed Project Characteristics and their influences and effects on Project 1

Project Characteristics	**Project 1 Indicators**	**Apparent Effects**
Commercial constraints	• Project to be delivered to given time schedule, to budget and be of a certain quality (TBQ)	• Estimating design and implementation effort was based on many unknowns • Specifying quality requirements concretely proved difficult and hard to measure against • Given commercial constraints and traditional lifecycle, a user-centred approach HCI techniques are hard to apply
Complex System	• Two people understood the complex KBS core of the software • Inherent complexity of GUI environment, which gives users many degrees of freedom	• Elements within the development team become autonomous and separate. • GUI event driven testing and error handling proved particular problems. • The event driven and object-oriented principles, fundamental to the Windows' GUI environment provided a paradigm shift of approach from structured programming**.**
Innovative System	• Designers and programmers have not seen a system which is similar to the one being constructed. Therefore initial expectations of what will be produced are unformed and mental models must be constructed from scratch.	• The lack of any initial building blocks for individuals mental models of the proposed system complicates team communication and focus. • The innovative nature of the Seed also contributed to the inherent complexity of the project.
Size of the project (people, budget, time)	• 8-10 people • £250,000 budget • 1 years duration	• This was a large project with 40-50% of available company resources committed to it • Communication difficulties due to team size
Commercial significance of project	• The most significant software development ongoing at the time. • Project 1 was fundamental to the company's Process Industry business strategy	• The opportunity afforded by the Project 1 software for the Sales and Technical areas was another possible factor which caused regular changes in requirements and objectives of the software.

1.3.1 Project Goals and Objectives

The goals and objectives of the Project 1 Seed were apparently not clear to all members of the development team. The ambiguity of the business and technical objectives was perhaps caused by there being no external client driving software requirements of the venture (although the technical manager was elected pseudo-client/user for the duration of the project). The speculative and innovative nature of the Seed project designed to meet a market need in what became a changing marketplace (suffering severely from the recession), also contributed to the confusion in project objectives, by generating new requirements throughout the duration of the project. One objective of the Seed project was to get a 'foot in the door' of a process engineering contracting company in order to improve the software company's process industry domain knowledge. The recognition of this need was somewhat double-edged, as the lack of domain knowledge at the outset of the project did not provide a good basis for specifying software requirements or objectives and also led to changes in requirements throughout the project duration (as domain knowledge improved).

Ambiguity in objectives caused disagreements between senior technical and sales staff early in the project. As the disagreements were between staff who had some first hand domain knowledge, the rest of the team derived their understanding of the project objectives from factions in disagreement. The resulting effect on the development team was that there was little common understanding of precisely what the project objectives or requirements were. This confusion led to confused technical approaches to the software development. On one hand, the lifespan of the Seed was considered to be relatively short with a major software development following in its tracks. On the other hand, the lifespan was to be considered long but evolving based on the initial Seed code. These objectives clearly represented two distinct software approaches analogous to revolutionary and evolutionary development, which would require distinct approaches to the programming task. Questions which a software engineer would usually ask, had confused answers; such as, 'should the code elements be designed to be re-usable?'. In fact, it is usual good software engineering practice to design code elements to be re-usable but on the Seed development some programmers were explicitly told by management not to bother with that. This kind of instruction would imply a revolutionary development approach yet other instructions, like the need to make code easily maintainable, would appear to contradict this.

1.3.2 Skills, Experience and Competence of Team Members

The skills, experience and competence of team members appeared to be an influence on the Seed project. Most of the software designers and programmers had considerable skills and experience of computer programming. However, the relevance of the skills and experience is unclear as this technical core of the team were primarily conversant with structured and data driven design and programming, with character-based user interfaces. The technical core of the team had little knowledge or experience of the Microsoft Windows event driven graphical environment, Object-Oriented Design (OOD) techniques or Borland C++.

A secondary business objective of the Project 1 Seed was to improve upon the software company's technical skills base by embarking on a project requiring new technologies - it was important for the company to become good at building Windows based applications. It was management's recognition of new skills required for Windows applications that led to the recruitment of an HCI designer. However, the HCI designer had no commercial experience in the HCI design role on a project and only possessed an understanding of HCI and rudimentary skills from university courses. If HCI design is considered as a craft skill the inexperience of the designer and the lack of a mentor is likely to have had a significant impact on performance.

The degree of inexperience within the team, both of the technology required for the construction of the Seed system and the new design approach involving an HCI designer led to considerable unknowns at the early planning stage of the project. For example, software designers would estimate the effort required to produce a part of the Seed system without first considering the UI design. It eventually transpired that the UI to the system involved a highly significant proportion of the overall design and programming effort.

The lack of knowledge and understanding of the constraints (often tacit constraints e.g. style) imposed by the Windows environment, coupled with the new possibilities offered by the GUI, led to considerable disagreements within the team and significant re-work to produce an interaction style consistent with Windows. Unfortunately the power of the development tools (especially BC++) was such that there were few instances where Windows style and conventions actually ***needed*** to be adhered to for technical reasons. Therefore, as most constraints were tacit and some programmers were unfamiliar with Windows even as users, inconsistent styles of implementation were common.

The competence of team members was variable, as was the working style of programmers in particular, some of whom would adopt a highly structured approach to their work, whilst others would be completely unstructured (commonly known in the software domain as 'Hackers'). Competence of individuals may be considered somewhat subjective and could be considered in the light of organisation culture. For example, in the software company, programmers were expected to gain some degree of domain understanding in order to evaluate their own work and make an effective contribution to the team. A programmer who fails in this regard could be seen as incompetent in the cultural sense but could still be extremely skilled at the raw programming task.

1.3.3 Team Experience of Working Together

The degree to which the team had worked together before appeared to influence the development of the Seed system. The software designers and programmers (the technical core of the team) and the project manager had worked together prior to the Seed project for between two and four years. The knowledge engineer and the HCI designer were new to the team and both introduced new roles. These roles and their responsibilities were allowed to evolve during the course of the project.

Changes in roles and responsibilities at times created tension and despondency within the team. One example of this was that some of the programmers used to enjoy designing the UI to their module of code, now this responsibility was given to the HCI designer. A more fundamental problem may be illustrated by a further example. The software designers saw the HCI designer as a solely a screen designer, and were unhappy about the HCI designer being involved in specifying the structure of the software or anything else which was not directly related to screen layout[1]. Changes in emphasis of the driving force behind the company's software developments, from a technically-driven to user-centred design approaches also changed responsibilities within the team, as the HCI designer was continually able to argue design decisions from a user perspective. The result of the changing roles and responsibilities was that team cohesion was poor and a 'them and us' situation developed between the technical core and more user-oriented team members (especially the HCI designer).

The inexperience of the team of working together affected communication within the team for two key reason. Firstly, because several team members had not worked together before, did not know what to expect from each other and had not attuned to each others abilities. Secondly, because of the tension and resistance to the changing roles and responsibilities of members of the team.

1.3.4 Team members' Understanding of the Domain

The software company's software development culture demanded that all team members have some understanding of the domain that the software is being developed for. The level of domain knowledge generally held by the team and the distribution of this knowledge throughout the team appeared to be an influence to the Seed system development.

As has been discussed, at the outset of the project the Seed team did not have an in-depth knowledge of the domain and this meant that requirements specified at the outset of the project ultimately changed as domain knowledge improved. As knowledge engineers progressed through the knowledge elicitation activity, their domain knowledge improved, as did that of the salesman through continued exposure to potential clients.

[1] The technical core of the team had historically worked mainly with character-based interfaces with limited scope for interaction and this had shaped their understanding of the UI. Initially, they failed to see that the new Windows environment had a great richness of possibilities for interaction which would place greater demands on the UI and necessitate an HCI design role on the team.

However, the distribution of the domain knowledge held by the team was poor as can be seen in the illustrations in figures B.1 and B.2. Domain knowledge was sought from the knowledge engineers and salesman by both the HCI designer and the project manager in order to perform effectively in their roles (figure B.1). Software designers and programmers rarely sought to gain any domain understanding as they saw this beyond the scope of their role, preferring to be told precisely what to do (in a technical sense and usually by means of a specification document) by other team members, thus rendering themselves unable to validate their own work with domain understanding (figure B.2). It should also be noted from figure B.2 that the domain knowledge of the software designers and programmers is overlapping, i.e. some programmers have domain knowledge which some software designers do not have and vice versa. This was perhaps due to individual performance differences or more likely, the fact that members of the technical core of the team would glean some domain knowledge from the particular aspect of the software they were working on.

Poor distribution of domain knowledge created a barrier to the formation of a cohesive team understanding of the objectives of the software and its context of use, i.e. there were difficulties in aligning the individuals' mental models. Without this alignment in understanding, the tasks of the software designer and the programmer needed to be clearly outlined in some form of precise and detailed specification, which did not exist. The software company culture aimed to provide realistic specification documents which required team members to think through and contribute to the specification issues themselves, thereby allowing the specification to grow and change with new insights and perspectives. Poorly distributed domain understanding created the need for a highly detailed specification, which was not effectively facilitated by the software company's development culture. The effect of poorly distributed domain knowledge and therefore unaligned mental models was lengthy project meetings (often with no concrete conclusions), many misunderstandings, heated discussions and frustratingly simple explanations to team members who had little domain knowledge.

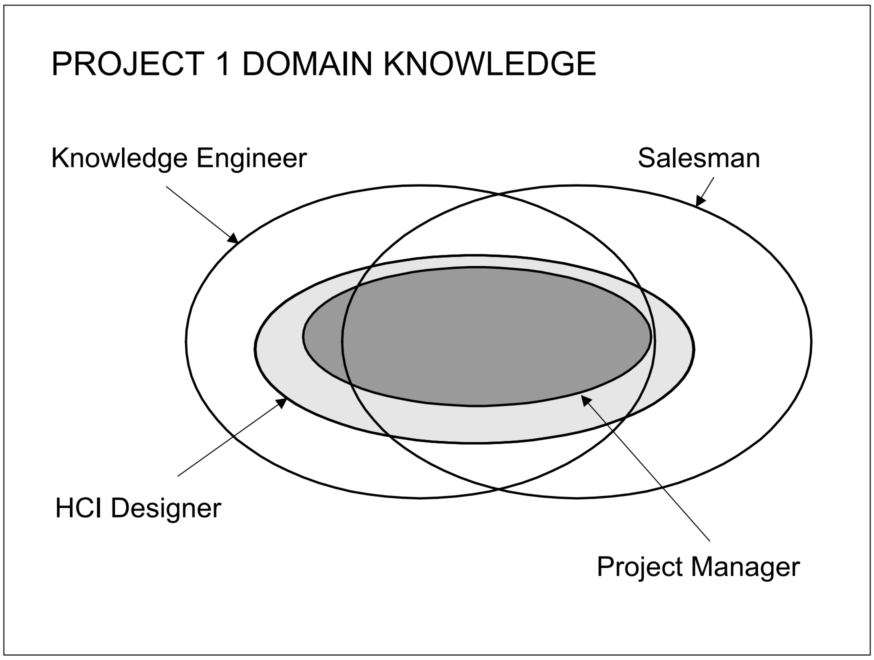


Figure B.1 Illustration of the Project 1 Seed domain knowledge distribution focussing on the HCI designer and project manager

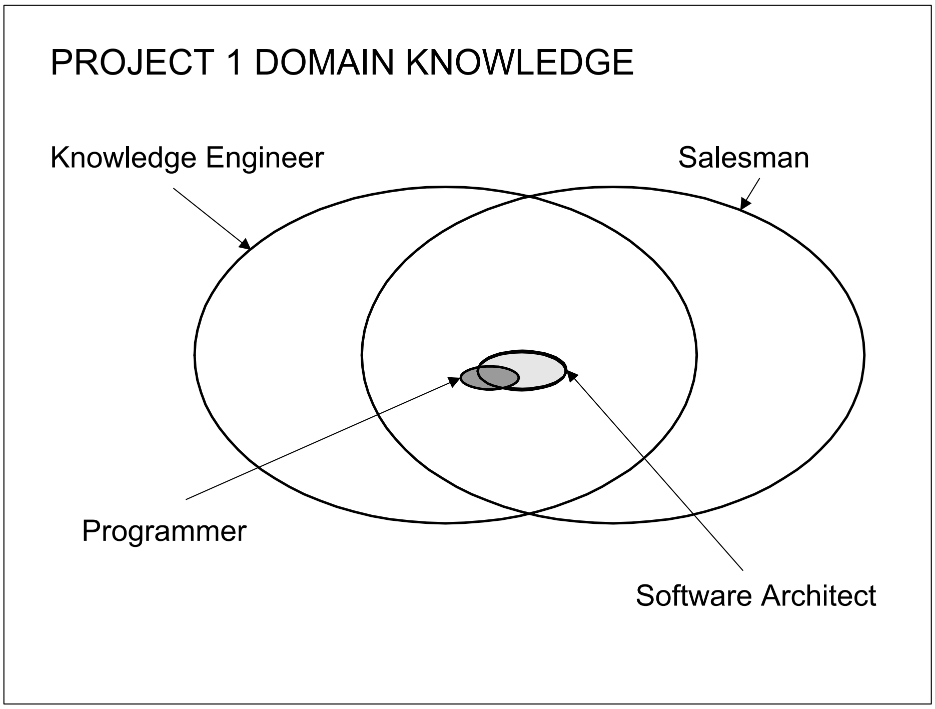


Figure B.2 Illustration of Project 1 Seed domain knowledge distribution focussing on the software designer and programmer

1.3.5 Commercial Constraints

Commercial constraints which demanded that the Project 1 Seed system be constructed in a given time period, to a given budget and be of suitable quality, had direct implications on the way the project was carried out. The need to estimate the effort required to construct the Seed was driven by the commercial need to examine the feasibility of and plan the project at the outset. However, with a considerable number of unknowns present, initial estimates were crude but the resulting budget was fixed. Software designers and programmers were always mindful of their initial estimate of the time required to complete a certain task. This meant that changes to the Specification or improvements that were required to the implementation would meet resistance as initial estimates may be exceeded. Increased quality arising from these changes was of minor importance to the technical core of the team, compared with staying within estimated budget.

A Gantt chart style project plan reinforced the waterfall lifecycle approach to the development and served to allocate tasks to individuals. Thus, the representation of the project plan itself emphasised performance of the individual over that of the team, and was linear so that a clear indication of progress could be given to management. Techniques used within the project which required iteration, in particular user-centred techniques, were complicated by the need to show progress in a linear way.

The commercial need to specify quality in the Specification (a contract) was not adequately dealt with in the Project 1 Seed Specification. For example, many possible and diverse user interfaces would have satisfied quality criteria in the Specification. The project manager of the Seed often asked, “What are we committed to produce?”, which actually meant interpreting the written specification in a minimalist way. Making effective statements about quality requirements was extremely difficult.

1.3.6 Project Complexity

The inherent complexity of the Seed software, both as a result of its KBS core and the GUI environment had an impact on the project. The complex core of the Seed meant that only two people had a good working knowledge of how the KBS part of the system functioned. Thus elements within the development team become autonomous and separate. Their activities became focused and the closed to the rest of the team. The complexity and exclusive nature of the KBS work limited the extent of the mental models of the system that could be constructed by other team members.

Further complexity was introduced through the selection of the Windows GUI platform, which necessitated the adoption of event driven and object-oriented principles, which proved to be a paradigm shift of approach from structured programming. The GUI environment created further difficulties due to the inherent complexities that arise from providing users with many degrees of freedom. The Windows environment also demanded a significant amount of effort to produce software which was consistent, operated in a compatible way with other software, looked good and felt slick.

Constructing the Seed to be sufficiently robust to meet these challenges and thoroughly testing the system, both became extremely difficult and lengthy activities. Constructing software to run on a Windows GUI platform clearly has inherent complexity.

1.3.7 Innovation

The innovative nature of the Seed project probably had its most significant effect on the team members' mental models. As designers, software designers and programmers had not seen a system similar to the Seed, initial expectations of what would be produced were vague, and mental models had to be largely constructed from scratch. This lack of any initial building blocks for individuals' mental models of the proposed system complicated team communication and focus, as was seen by the levels of misunderstanding that occurred. The innovative nature of the Seed also contributed to the inherent complexity of the project as considerable creativity was necessary to construct the software.

1.3.8 Project Size

The size of the project, in terms of budget, time and the number of people involved in it influenced the Seed project. The software company invested £250,000 in Project 1 and utilised 40-50% of staff resources for almost a year. From the perspective of the software company, the project was large but in general it might be considered a medium sized development. Of the size dimensions, the number of people in the project team had the most direct influence of the development. The team was made up of 8-10 diversely skilled individuals, several of them working together for the first time. This number of people in a team created communication difficulties and lack of visibility of each others work. The size of the weekly project meetings was deemed too cumbersome by the project manager who decided to split the meetings into three groups. This further isolated factions within the team and created a new barrier to visibility of each others work. The size of the team and the isolation of factions further contributed to a 'them and us' situation between the technical core of the team and the user-centred faction (primarily the HCI designer but also the project manager with his Quality Assurance responsibility). The size of the team also seemed to provide a barrier to effective team work, which was observed as poor team work. There are many examples of poor team work throughout the project, one example was when two programmers sitting next to each other, working on different elements of the Seed which would ultimately be joined together, made assumptions about each other's code rather than ask about it - resulting in considerable re-work.

1.3.9 Commercial Significance of the Project

Commercial significance and profile of the project had a possible influence on the development. Project 1 was the most significant software development ongoing in the software company at the time and management expectations were high. Project 1 was fundamental to the software company's Process Industry business strategy. The opportunity cost of utilising 40-50% of the software company's people resources was clearly significant. Thus, the profile of the project and high expectations probably increased external pressure on the team, possibly offset by a compassionate stance adopted by management towards the team. The significance of the Project 1 Seed opportunity for the sales team in particular, was a probable driver for continual reassessment and change of requirements and objectives for the software.

1.4 The Organisation

Characteristics of the software company in terms of organisational structure, culture and management appeared to have been an influence on the development of the Project 1 software. These characteristics and apparent effects are shown in table B.2 and are discussed in greater detail in the following sections.

Table B.2 Organisational elements and apparent effects on the Project 1 software development

Organisational Elements	Considerations	Apparent Effects
Object-oriented (matrix) organisational structure	• Designed for empowerment and sharing responsibility • No rigid definitions of roles or responsibilities	• Role ambiguity
Culture	• Designed to encourage staff 'buy-in', i.e. active contribution, including learning about the domain, to enable self evaluation of work • Emphasis on communication and teamwork • Specification is an important milestone, however the document is ostensibly open to change	• Software designers and programmers failed to gain much domain knowledge • Many software designers and programmers preferred to work alone and from specification documents • Specification also forms a contract, so there is certain resistance to changing it
Management understanding of programming	• Senior management and sales personnel had no first hand experience of programming • Project manager was formally a programmer (over ten years ago)	• Misconceptions about programming • Failure to realise paradigm shift from structured programming to object-oriented

1.4.1 Organisational Structure

The software company is structured as an Object-Oriented (matrix) organisation partly in order to empower employees and enable responsibility to be shared. The resulting culture did not encourage rigid definition of roles and responsibilities within the software team. The new HCI designer role and the particular skills of other new individuals were allowed to find their place in the team. One observed effect of this was that a certain amount of role ambiguity was apparent. For example, one of the programmers resented the loss of a creative outlet when he could no longer design the UI himself. A further example relates to numerous occasions when software designers and programmers were unhappy about an HCI designer showing concern about software structure (being trained in this area, the HCI designer was fully aware of the potential impact on the UI of an inappropriate software structure).

1.4.2 Software Development Culture

The software company's culture aimed to encourage its staff to "buy-in" to projects and each actively contribute by gaining some understanding of the problem and the overall solution concept. By this means team members should have always been in a position to evaluate their own work, rather than working unquestioningly from written specification documents. The full benefit of this sentiment was not realised during the project as the software designers and programmers gained little domain knowledge.

Strong emphasis was made on teamwork and communication (both formal and informal) in order to deal with the dynamic and complex nature of the software being produced. However, many software designers and programmers preferred to work alone and from written documents.

Whilst emphasis was placed on the Specification document as a key milestone in the project, it was understood that information in the Specification could be questioned and changed. However, as the Specification document was signed-off, it was in the project manager's interests to ensure that no further changes were made as the Specification formed a contract outlining what would be produced by the venture.

1.4.3 Management Understanding of Programming

The software company's senior management and sales personnel had little first hand knowledge of the technical programming task. However, the project manager was formally a programmer, albeit in a quite different environment and over 10 years ago. Misconceptions about the programming task appeared to be held by management; for example, programmers were apparently told NOT to design re-usable code elements, which is often considered basic good practice for programmers. Management also expected software designers and programmers to train themselves on new approaches required by the new tools being used on the Seed. However, the magnitude of transition from structural data driven programming to object-oriented event driven programming (effectively a paradigm shift) was not recognised by management. This is not surprising as it is usual to expect a programmer to learn a new development language or environment quickly and by teaching themselves.

1.5 The Team

The cultural emphasis on the new team, combined with the diverse skills and backgrounds of team members and influences attributable to the commercial setting, appear to have been an influence in the software development. Table B.3 summarises team considerations and the points raised are described in more detail in the following sections.

Table B.3 Team considerations, observations and apparent effects on the software development

Team considerations	**Observations**	**Apparent Effects**
Cultural emphasis on team and reduced emphasis on written documentation, e.g. Specifications	• Traditionally, technical core of team worked unquestioningly from written specifications, in segmented groups with considerable independence . • Low needs for social contact	• Team did not perform as a cohesive unit
Diversity of skills and backgrounds evident in the new team	• Ambiguity of roles and responsibility • Respect between disciplines lacking • Only the technical core had worked together before • Most of the team were unfamiliar with the new roles • Senior team members fearful of (threatened by) younger ones in the new team	• non-cohesive team • 'Them and us' situation evolved • lack of familiarity brought communication difficulties • fear was detrimental to team, especially fear of making mistakes and attribution of blame for mistakes that occurred
The influence of commercial realities	• The project meetings were segregated into three disciplinary areas for convenience • Team composition changed through the duration of the project	• Segregation created further barriers to communication • Loss of knowledge and experience from the team • Need to instruct and train new team members

1.5.1 Cultural Emphasis on Team

In acknowledgement of the complexity of software that the software company constructs and the perceived highly skilled nature of their personnel, the development culture gave strong emphasis to the skills and knowledge of the team, over that of written documentation. To produce a detailed written Specification of the software at the outset of the project was deemed infeasible due to an acknowledged lack of domain knowledge and the practical consideration that a detailed written Specification document would be huge. The intention was that the Specification could be minimised given that team members were to have a considerable understanding of the problem being addressed and the solution offered by the Project 1 software. The development team did not fulfil these expectations. A possible reason for this may be attributable to the traditional working practices of the technical core of the team. Historically, the technical core of software designers and programmers worked in segmented groups from extremely detailed specification documents (often being several volumes long). These people had worked with considerable independence and had been trained to unquestioningly construct code from detailed written requirements, specifying outputs to given inputs. The personalities of most of the technical core of the team appeared to have low needs for social contact in the workplace and so preferred the traditional approach to their work. Thus, the team oriented approach demanded by the software company to produce the Project 1 Seed did not suit all of these people, many of whom would often comment that they were being asked to do something which was, “not in the Spec”. Therefore, the Project 1 team did not operate as a cohesive unit.

1.5.2 Diverse skills and backgrounds

The diversity of skills and backgrounds evident in the software company’s new software team is also likely to have influenced the software development.

Ambiguity of roles and responsibilities within the team coupled with a lack of respect between various disciplines did nothing to help the team cohesivity. For example, the programming task was considered trivial by some people; whilst others would disrespect the HCI activity and the importance of user considerations. Programmers would often argue against a UI enhancement by stating that they considered it more important that the system was able to run properly. It was then not surprising that a somewhat ‘them and us’ situation evolved between the technical core of the team and the user oriented areas.

Only the technical core of the Seed team had worked together before and new roles, skills and diverse disciplinary backgrounds of new team members introduced a certain degree of unfamiliarity. The team were largely unfamiliar with each other as well as the new roles that had been introduced. The lack of familiarity brought with it communication difficulties within the now multi-disciplinary team. One example was the continual failure of hard-nosed technical individuals to appreciate the impact of UI bugs on the user.

Some of the previously senior members of the technical core of the team seemed to be fearful of the younger and more dynamic team members and perhaps even saw them as a threat to their own job security. An example of such fear could be seen by the reaction of software designers and programmers when bugs were found in the code. These technical professionals looked upon each bug as a potential personal affront reflecting on their abilities. Therefore, considerable effort was utilised in the blaming and finger-pointing processes as the fear of being wrong was serious. Such fears and associated behaviour presented barriers to team bonding and were nothing but detrimental to the team performance.

1.5.3 The Impact of Commercial Realities

Further barriers to communication and team cohesion could be attributed to the project manager who decided that weekly progress meetings had become too unwieldy, so segregated what was the only common forum for the team into three sessions..

Resource availability demanded that during the course of the project, the composition of the team and the roles of team members had to change. Initially, the project team consisted of a technical manager, an HCI designer and a knowledge engineer. Ultimately, the core project team comprised a project manager, a technical manager, an HCI designer, a knowledge engineer, three software designers, and three programmers. During the course of the project, several people were to leave the team, examples, included a software designer. Other team members were to find their roles change, in particular the HCI designer who became more involved in Quality Assurance (QA) and the technical manager who became a pseudo-client/user. The main effect of such changes was that some knowledge and experience was lost from the team, and new team members (often programmers) needed instruction and training.

1.6 Representations and Techniques

The selection, availability, familiarity and use of representations and design techniques appeared to influence the software development. In particular, task analysis, the internal design of the software, the HCI design, prototyping and motivations for OO design were regarded as particularly important. Table B.4 summarises the issues which are discussed further in the following sections.

Table B.4 Representation issues, considerations and apparent effects

Representation issues	**Considerations**	**Apparent Effects**
Task Analysis	• Task models produced for the Demonstrator project were hard to interpret as they were not self explanatory and had little supporting documentation. • Many suggested task analytical techniques suggested in literature were not relevant to the Seed development. This is because there was no direct access to end-users and task information had to be pieced together from what was known within the team. • The idea of a **Future Task Model (FTM)** was devised by the HCI designer in response to a need articulated management. This diagram showed how the user's tasks would be structured following the introduction of the Seed system.	• Hierarchial Task Analysis (HTA) was the selected method of representing the current user tasks. These diagrams were more self explanatory and allowed previous task analyses to be incorporated in the diagrams. • The FTM showed team members, management and sales people exactly how the Seed system would integrate with the user's tasks and how those tasks would be changed. • The FTM was undertaken after a good understanding of the domain, current users and the proposed Seed system had been gained by the HCI designer. • By improving the HCI designer's mental model of the Seed software concepts, the FTM enabled the design to be refined. Thus, production of the FTM and the Seed software concepts was an iterative process.
Internal Design of the Software	• Technical core were familiar with representations and techniques used for structured programming • Structured programming design representations and techniques appeared inappropriate for Windows and event driven code. • OOD representations and techniques proved hard to learn and inaccessible.	• The traditional DFD representation was adapted for Windows use • Familiarity with DFDs rather than appropriateness for the design task dictated its adoption
HCI design	• OOD literature failed to explain how OO representations could be used for the representation of the UI • HCI representation of the Seed ultimately took the form of screen ERDs, sketches, walk-throughs and visual prototypes	• The HCI designer adapted a familiar representation (ERDs) to represent the Seed • The HCI designer's limited sketching ability hampered the use of hand-drawn representations, resulting in an early move to use computer tools

Table B.4' continued Representation issues, considerations and apparent effects

Representation issues	**Considerations**	**Apparent Effects**
Motivations for using OOD representations	• The development language and environment were OO, so an OOD seemed appropriate and necessary • Perceived need of the HCI designer to integrate UI and HLD by having consistent representations • The software company's strong emphasis on the OO philosophy, going even as far as company structure	• There was not an easy transition to OOD representations and techniques, although they did appear to be beneficial for use with C++ and Windows. • Integration of OOD and HCI representations was not direct. HCI representations such as sketches, ERDs and the prototype offered an alternative view of the Seed system than was offered by the OO and DFD representations.
Prototyping	• Technical core of the team resisted looking at the prototype. • Computer prototyping tools enabled an experimental approach to be taken towards HCI design. • A major benefit of prototyping was increased visualisation	• There was limited access to the prototype in that it was only available on the HCI designer's computer • Prototyping was a new technique not previously used in the software company's developments • The technical core may have felt threatened by the fact that it was easier to produce apparent functionality in the prototyping tool (VB) than in the development language (BC++) • Code elements in the development language often produced a UI which looked to be of lower quality than the that in the prototype. Thus, expectations raised by the prototype were often not met by the final system. • Visualisation helped the HCI designer to generate and test ideas. • Visualisation afforded by the computer prototype aided the communication of design intent more effectively than static representations like sketches.

1.6.1 Task Analysis and Future Task Model (FTM)

Task models produced for the Demonstrator project were hard to interpret as they were not self explanatory and had little supporting documentation. Therefore, a brief examination of available literature was made in order to find a better representation. Many suggested task analytical techniques suggested were not relevant to the Seed project which had no direct access to end-users. Task information was pieced together from the software company's domain 'experts' within the development team.

Hierarchial Task Analysis (HTA) was the selected method of representing the current user tasks. These diagrams were relatively self explanatory and allowed task analysis carried out for the Project 1 Demonstrator to be incorporated into the diagrams (see Appendix A figure A.2).

The idea of a **Future Task Model (FTM)** was devised by the HCI designer in response to a project need which was reinforced by management observations. The evolution of the need is described in the April 1993 section of Appendix A as follows,

"*During the exercise of writing the Specification it became apparent that team members were still unclear how the Seed would look and behave. Even though a partial prototype existed and by now several iterations of each screen design had been achieved, it was still difficult to visualise the Seed system in use. It was felt that the task models should have solved this problem but in their existing format did not provide what was required. As the HCI designer I was charged with looking at how the current conceptual design would facilitate the engineer's tasks. The approach taken to address this problem, was to produce a model of how the engineer's task would be structured if the Seed system were in place, this was called a **Future Task Model**.*"

Thus, the FTM was created to showed how the user's tasks would be structured following the introduction of the Seed system.

The FTM proved particularly effective at helping management and sales people and some team members (those with little domain knowledge) visualise exactly how the Seed system would integrate with the user's tasks and how those tasks would be changed.

The FTM comprised several basic scenarios, a summary of the main tasks necessary to achieve each scenario, followed by a detailed breakdown explaining how the user would proceed through the main tasks using the Seed system. Figures A.6 and A.7 in Appendix A. show actual examples of the FTM constructed.

The method of constructing the FTM could be considered craft-based as the model was formed solely from the HCI designer's mental models of the user's current tasks and the proposed Seed system. This too can be seen from direct reference to the April 1993 narrative of Appendix A:

"*The FTM was 13 pages long and produced in one afternoon based on my mental model of the Seed system and my understanding of the engineer's tasks.*"

Understandably, the FTM in its final form was not completed until the HCI designer had gained a good understanding of the domain, current users and the proposed Seed system. In fact, by improving the HCI designer's mental model of the Seed software concepts, the FTM also enabled the design to be refined. Thus, production of the FTM and the Seed software concepts was an iterative process. A simplified conceptual representation of how the FTM was constructed can be seen in figure B.3.

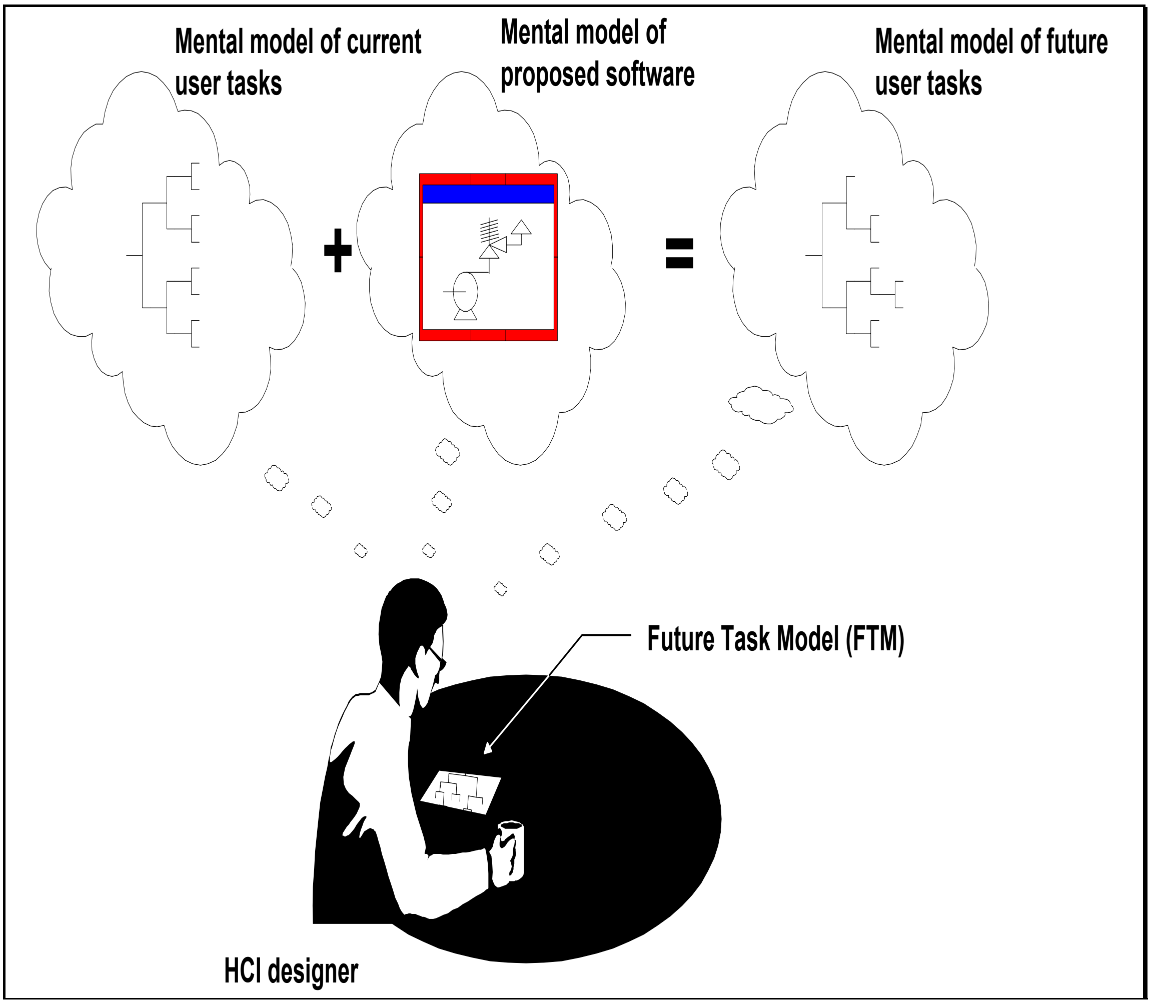

Mental model of current user tasks
Mental model of proposed software
Mental model of future user tasks
+
=
Future Task Model (FTM)
HCI designer

Figure B.3 A simplified illustration of the construction of the Future Task Model

1.6.2 Internal Design of the Software

It was felt that an OO approach should be used for the internal design of the Project 1 software. However, the technical core of the team were familiar with techniques and representations that have been used for the design of structured programming projects for many years but these proved inappropriate for the Windows event driven OO architecture. Following an examination of object-oriented design (OOD) representations and techniques it was evident that the OO paradigm was not easy to learn, nor was it particularly accessible (there were many conveniently simple examples in available texts). Therefore, the old Data Flow Diagram (DFD) representation was deemed still useful in an adapted form. It was more likely the familiarity with the DFD notation rather than its effectiveness as a representation which led to its adoption.

1.6.3 HCI Design

The HCI designer also investigated OOD notations as a means of representing the UI. Investigation and previous experience indicated that OOD should be an effective means of representing a UI but examples in available literature failed to address UI needs. Therefore, like the technical core, the HCI designer also adapted a familiar representation called Entity-Relationship Diagrams (ERDs) to represent the structure of the Seed system. It transpired that at a basic level, ERDs and OO Class diagrams were not too dissimilar.

Representation of the Seed system from an HCI perspective was eventually conveyed primarily by screen sketches, walk-throughs and visual prototypes. The sketching ability of the HCI designer hampered the use of hand-drawn pencil sketches as a representation. This resulted in an early move to using basic computer tools to paint screens. Ultimately, for the design of the results module - the most complex element of the Seed system - a software prototype was constructed.

1.6.4 Motivations for using OOD representations

There were several motivations for the technical core of the team and the HCI designer to find appropriate OO representations and techniques. Firstly, the development environment and language were OO so an OOD seemed appropriate and necessary. Secondly, there was a perceived need by the HCI designer to integrate the UI design and the High-Level Design (HLD) of the software. Therefore, consistency of representations used by the technical core and the HCI designer were seen as important to effective communication within the team. A third motivation for the shift to OOD representations came from the software company’s strong emphasis on the OO philosophy in general, to the extent of having an OO company structure.

Ultimately, direct integration of HCI and internal design representations was not achieved, rather HCI representations such as sketches or the prototype, offered an alternative view of the Seed system than was seen in the OOD or DFD representations. The transition to an OO approach was not easy during the Seed project but enough was learnt to realise the usefulness of the approach to developing event driven Windows software.

1.6.5 Prototyping

Following the construction of the prototype of the results module, it became apparent that members of the technical core of the team were resistant to looking at the prototype. This resulted in it only being used in formal review sessions or if activated by the HCI designer to explain a particular element of desired functionality. There are several possible explanations for this distinct resistance to looking at the prototype, the simplest of which is that it was only available on the HCI designer's computer. A more likely explanation is that prototyping was an unfamiliar new technique not previously used in the software company software developments. A further credible explanation is that the prototyping tool (Visual Basic) enabled the HCI designer to quickly mock-up apparently functional Windows software. It is likely that the technical core of the team were somewhat threatened by this because using the development language (BC++) it was far more difficult to actually implement the apparent functionality of the prototype. In fact, after considerable effort it was usually the case that the actual implementation did not look as good as the apparent functionality represented in the prototype. This caused further problems for the technical core of the team as the prototype had raised expectations of what could be produced in Windows, and they were expected to construct the software for real. The most dangerous of these expectations were held by management who did not fully appreciate the considerable difficulty of the programming tasks involved in the construction of the Seed.

Notwithstanding the reluctance of the technical core of the team to look at the prototype, the technique did appear to show great potential. As a tool for HCI design, prototyping allowed an experimental approach to design ideas which could be quickly tested and animated in a realistic Windows style of interaction. A benefit of this approach was the aided visualisation of the design that the tool could provide. This visualisation was not only useful to the designer's experimentation but also aided the communication of design intent to other team members. Members of the technical core of the team could more easily appreciate design intent conveyed from an animated prototype than from static pencil sketches. Thus the creative leap typically necessary in interpreting sketches, was reduced when using a realistic software prototype.

1.7 The Specification

Issues surrounding the specification document appeared to be an influence on the Seed development. In particular, the diversity of uses and users of the specification and the complexity of the authoring task were problematic. Commercial realities, such as the need to plan and estimate effort early on in the process do not provide a useful means of dealing with technical unknowns. The development culture emphasises the team over the written documentation, which should facilitate changes as team members see fit but the development process and the fact that the specification is a form of contract, conspire against the intended culture. Table B.5 summarises the issues which are discussed further in the following sections.

1.7.1 Development Culture

The software company acknowledged that when producing complex and innovative software like the Seed system, for a relatively unknown domain, it was infeasible to adequately specify the proposed software in great detail during the early stages of the project. The software company understood that its development team needed time to familiarise themselves with the new domain and concepts for the proposed software.

Producing the specification for the Seed system was nonetheless a major undertaking bringing together work from many areas of the project, for example, client visits by knowledge engineers and requirements analysts, investigation into the domain, prototypes to test proposed software concepts and future tasks of users, preliminary software internal design, etc. The fact remained however, that the specification was produced during the relatively early stages of the project but was theoretically open to question and change by team members throughout the duration of the project. This level of flexibility is key to the software company's software development philosophy and culture.

Not only is the specification left open to change, it is rather the case that the specification was not expected to be completely right first time and would have to change as the team gained experience. The specification itself was only around 100 pages long which was rather short compared with conventional software specifications. The software company expected its software team, in particular software designers, programmers and HCI designers to work together to fill in the details and make necessary changes as the project progressed

Table B.5 Specification issues, considerations and apparent effects

Specification Issues	**Considerations**	**Apparent Effects**
Development Culture	• The software company recognised that producing full, complete and detailed specifications early on in the development was infeasible. • The software company culture relied on the team to question and change the specification as the project progressed, using their skill and professional judgement.	• Producing the initial specification still involved considerable effort, e.g. contact with domain experts, task modelling, knowledge modelling, prototyping, internal design.
Commercial Realities	• The specification was divided into parts which could be programmed by an individual. The individual then estimated how long it would take to program and this was entered into the project plan. • The specification was produced when a general understanding of the problem and the proposed solution had been attained. • The changing composition of the team puts a communication burden on the specification document.	• Programmers were motivated to deliver code within their initial estimates and so resisted changes. • Early estimates were made while there were still many technical unknowns. • The process did not motivate programmers to be concerned about Quality, only Time. • At the outset of the project the project manager had to predict when the Specification would be produced as this was a key project milestone • Within the specification document, it was simple to specify Time and Budget constraints but Quality was extremely difficult to specify.
Users and uses of the specification	• There were diverse users and uses of the specification. • The level of detail in the specification was not geared towards the skill level and cohesiveness of the team	• The use of the specification as a contract appeared to be a serious flaw in the approach. • The mixed abilities and backgrounds of the specification readership, and the diverse uses of the document were not addressed by a low quality and poorly written specification. • Different representations used within the specification gave different views on the software. The prototype was probably the easiest view to comprehend. • Computer prototypes were used to animate the specification to aid communication.

1.7.2 Commercial Realities

For implementation, the specification was divided into parts which could be programmed by an individual. Each individual then estimated how long it would take to program and this was entered into the project plan. These early estimates were founded on many unknowns, particularly because of the team's lack of familiarity with C++, Windows, Object-Oriented principles, the complexity of the KBS and HCI design. However, technical people within the team were apparently strongly motivated to deliver their part of the code module within their initial time estimates. Therefore, any changes that were required to a module were fiercely resisted by the technical person responsible for it, fearing that the changes would cause them to exceed their initial estimates. In fact, any changes soon became met by a negative reaction (almost a reflex) on the part of some of the technical people. For example, when one aspect of the functionality of a module was significantly reduced, SD/P_1 sucked his teeth and said, "I'll have to revise my estimates" (implying that it would take longer to implement with these new changes), when in fact, the cut in functionality clearly reduced the work to be done.

The development process did not motivate programmers and other technical staff to be concerned with the quality of what they are producing, only the time it takes to produce it. Therefore, the commercial need to plan and deliver the software on time motivates programmers to resist changes to the specification.

A further impact of commercial realities is the project manager's need to plan. Theoretically, the specification should have been produced when a general consensus indicated that a good understanding of the problem and the proposed solution had been attained. However, at the outset of the project, commercial necessity dictated that the project manager must estimate when this consensus would be reached, as the specification was a key milestone on the project plan.

In order to write the Specification, the software team comprised the technical manager, knowledge engineer, software designers and the HCI designer. Following the production of the Specification, the technical manager and a software designer left the team and several programmers joined it. The loss of expertise in the team and addition of new members created a burden of communication on the Specification, which had to on one hand, capture lost expertise and on the other, inform the new team members. The Seed Specification did not perform well in this role due to the poor quality of the document. The combined knowledge of team members remaining on the team throughout the project and the software prototype, were probably as important as the Specification document in transferring information to new team members.

As will be discussed in the following section, the specification formed a kind of contract for what the software would deliver, when and for how much. Despite difficulties in arriving at a budget and deadline for the project, it is nonetheless a simple matter to state them when they were agreed. It was decidedly more difficult to precisely specify quality and HCI requirements for the Seed system. Therefore, vague statements made about quality and HCI in the specification of the Seed could have been satisfied by any number of user interfaces. In other words, quality and HCI statements did not specify strong constraints on the development.

1.7.3 Users and Uses of the Specification

Table B.6 illustrates the users and uses of the specification document. From the table it can be seen that in addition to the multi-disciplinary nature of the software team, managers and the pseudo-client/user, each with different backgrounds and perspectives needed to understand the specification at some level. The problem of diversely skilled readership was addressed by using different representations within the specification to give different views on the software. Representations included descriptions of end-user tasks and their likely future tasks, technical object-oriented design representations (e.g. class diagrams), data flow diagrams, etc.. In addition to the written specification, a software prototype was produced to help a variety of people to visualise the complicated results module of the Seed. The prototype became a kind of animated specification, which in itself was the cause of some difficulties within the development team. The prototype was produced by the HCI designer and concentrated on the user interface to the software. Many aspects of the required interaction, quality and style of the software are demonstrated in the prototype more effectively than in the written specification. However, the effectiveness of accurately conveying the HCI design intent through an animated prototype appeared to depend upon how members of the multi-disciplinary development team interpreted what they saw in the visualisation. Technical team members appeared to derive information about what data was displayed whilst failing to recognise what may be complex or troublesome user interface mechanisms. The prototype did not clearly show which bits of the visualisation were important and which were not. Unlike the written specification which is often viewed as a form of contract, features shown in the prototype were apparently not considered contractually binding. HCI and quality features implied in the prototype were easily dismissed within the commercially oriented development context.

Ultimately, the use of the specification as a contract proved to be a serious flaw in the flexible approach taken to specification. Table B.6 shows that the specification acted as a contract between the software company management and the project manager. It was the project manager's motivations and direct influence in particular that contradicted the openly flexible approach to specification. The project manager was committed to delivering software which met the specification in terms of quality, budget and timescales. However, it was much more difficult to specify quality than it was to specify budget and duration. Many statements relating to quality or usability were general or subjective, which made cutting corners in these area much easier than going over budget or missing a deadline. Therefore, it was generally not in the interests of the project manager to make changes to the specification, particularly if they were likely to involve more work than was initially planned. Therefore in general, when discussing changes to the specification, the project manager would usually put forward a minimalist view, often asking, "what are we committed to produce?", which meant, what does the letter of the specification say? Although the project manager was ultimately responsible for the software quality and therefore did have an open attitude towards changes (and many were made to improve quality), he was probably more strongly motivated by time and cost which were more concrete constraints.

Table B.6 Users and uses of the Seed specification document

User	Use
Project Manager	As a ***contract*** of what would be delivered to the client (in this case pseudo-client) in terms of functionality and quality. Also defining the project budget and the delivery deadline. As with other team members, the project manager also uses the specification to help construct a mental model of the proposed software.
Members of the development multi-disciplinary team	As a working document specifying functionality of the proposed software. i.e. capturing and conveying a conceptual model of the proposed software.
Senior software company management	As an illustration of what the proposed system will do and how it will meet the user requirements. This was sufficiently detailed for management to accept the software venture and sign-off the project budget. It also formed a ***contract*** of what would be delivered to the client, through the project manager in terms of functionality and quality. Also documenting the project budget and the delivery deadline.
Pseudo Client/User	As an illustration of what the proposed system would do and how it would meet client requirements. From a user perspective, it indicated how the proposed software would facilitate the users' tasks in the future.

The diverse readership and uses of the specification document would suggest that the authoring task was difficult. In fact, little consideration was given to the authoring of the document or its resulting quality. In fact, the quality of the Specification document for the Seed project was extremely low. Editing had consisted of gathering together reports written by team members and appending a front cover. The document contained several different fonts and writing styles, and was difficult to read.

The level of detail in the Seed Specification was apparently insufficiently aligned with the skill level and cohesiveness of the software team. For example, most of the programmers were unfamiliar with programming Windows and so had only a basic understanding of Windows applications. As such, the specification document should perhaps have included more detailed explanations about Windows functionality and structure than would be necessary in a specification for experience Windows developers.

1.7.4 Conclusion

The flexible approach taken towards specification and emphasis placed on the skills of the development team to fill in the details as the project progressed, in theory, appeared to be a sensible approach. However, commercial realities demanding that the specification form a kind of contract, motivated the project manager to resist changes in order to deliver the software on time and under budget, thus contradicting the desire for flexibility. Similar commercial constraints and resistance to changes applied to technical team members, such as programmers, who were strongly motivated towards delivering their individual code modules within initial time estimates they had made. This meant that quality and HCI requirements which were not adequately specified in the written specification and were only implied by the prototype were open to interpretation. Commercial realities also dictated that the composition of the team change at the specification milestone which made the specification document and the associated computer prototype a kind of communication focal point in the development process. Further commercial demands necessitated multiple uses for the specification and a diverse readership. The limited prototype was used to animate the results module from the specification and convey design intent to a variety of people. However, the way the information represented in the prototype was interpreted appeared to depend on the individual's background, motivations and perception. Basic commercial needs like the need to plan a project at the outset, whilst there are many unknowns, make even the timing of the specification difficult. Whilst the flexible approach to specification helped to address the reality that the specification would need to change, basic commercial pressures, such as the use of the specification as a contract influenced the team to resist changes.

1.8 The Lifecycle

Table B.7 summarises some considerations of the lifecycle adopted for the Seed system, characteristics of the approach and apparent effects.

Table B.7 Lifecycle considerations, observed characteristics and effects

Lifecycle considerations	**Observed characteristics and apparent effects**
The Waterfall approach	• The waterfall approach is inherently not iterative and therefore not user-centred. • Some iterative activities took place within certain aspects of the development, e.g. prototyping the results module. • The adoption of the waterfall lifecycle was perhaps due to the fact that the project manager and technical team members were familiar with it and had always worked that way • The project plan, represented by a Gantt chart and being based on the waterfall lifecycle, was useful for monitoring progress through linear phases. • The lifecycle did not provide a clear framework for the team to work together within. Ambiguity of roles and responsibilities were not clarified by the life-cycle.
HCI design	• The UI development plan was not integrated with the main development lifecycle until the latter stages of the project. • HCI design became a bottleneck, reducing the time available for programming • HCI design activity was on the critical path of the project, contributing to delays and causing tension in the team and stress centred on the HCI design who felt obliged to put in long hours.

1.8.1 The Waterfall Approach

The software lifecycle used for the development of the Seed system was loosely based on a traditional waterfall model. This is a linear lifecycle which fits a waterfall analogy as activities and phases progress step-wise towards the project's end with no iteration between activities, i.e. in the same way that water cannot flow up a waterfall. As the waterfall approach is inherently not iterative, it could also be considered inherently not user-centred, although iteration can occur within activities. In fact, iterative activities did take place during the Seed project, e.g. prototyping the results module.

The adoption of the waterfall lifecycle was perhaps due to the fact that the project manager and technical team members were familiar with it and had always worked that way. However, the approach does have some features which are clearly useful for commercial software development. For example, the project plan, represented by a Gantt chart and based on the waterfall lifecycle, was useful for monitoring progress through linear phases.

A major drawback of the waterfall approach adopted was that it did not provide a clear framework for the team to work together within. Ambiguity of roles and responsibilities were not clarified by the lifecycle.

1.8.2 HCI design

The UI development plan was not explicitly integrated with the main development lifecycle. However, during the latter stages of the project, HCI became a key part of the main development plan.

Perhaps one of the reasons for the integration and raised status of the HCI activity was that it was soon seen to be a bottleneck in the process. Ultimately the HCI detailed design activity had up to five programmers awaiting designs before they could progress. The HCI design activity was also late finishing and directly reduced the amount of time available for programming.

The HCI design activity was on the critical path of the project for a significant period of time during the design phase. This caused excessive tension in the team and stress focused on the HCI designer who then felt obliged to put in extremely long hours.

1.9 Characteristics of Individuals

Characteristics of individuals within the development team appeared to have significant influence on the project. Table B.8 summarises key characteristics of individuals within the development team.

Table B.8 Observed Characteristics of individual team members and their influences and effects on the Project 1 software

Individual Characteristic	**Project 1 Indicators**
Skills and experience	• inexperience of HCI designer, lack of sketching skills and inaccessibility of HCI theory • consideration of skills as craft-based • lack of skills relating to OOD, C++, Windows, GUI and event driven programming • relevance of programming skills and experience • experience of management and understanding of programming task • technical core of the team and project manager had no experience of interacting with new roles, e.g. HCI designer
Competence	• general competence at performance of task • competence relating to organisational cultural expectations
Attitude	• traditional software engineering values cause resentful attitude towards new culture and ideas, e.g. specification • lack of inter-disciplinary respect • the technical core of the team are primarily concerned with specific functionality rather than user concerns • fear of appearing foolish in front of peers
Roles and Responsibilities	• new roles changes roles and responsibilities within the team • ambiguity of role and responsibility • general role diversity and multiple disciplines • individuals roles changed during the project • similar elements of project manager and HCI designer role • diversity of roles within the programming job
Motivation	• technical core are motivated to produce an elegant technical solution • HCI designer motivated towards producing a appropriate and effective UI • perception of technical core was attuned to functionality not interactivity
Personality	• able to work in a team • poor social and communication skills of technical core • different personalities skills and performance of technical core

1.9.1 Skills and Experience

The majority of team members on the project were lacking in skills or experience in areas fundamental to their roles.

The HCI designer had never before done any commercial HCI design but had participated in the production of commercial software in different roles. The HCI designer was further hampered by a lack of sketching ability which led to premature use of computer tools for representation of ideas. Furthermore, the general inaccessibility of practical HCI theory presented further barriers to performance. If HCI is considered a craft skill, on Project 1 the HCI designer should have been an apprentice but had no teacher.

Software designers and programmers had little skills or experience of object-oriented design (OOD), designing for GUIs or Windows, event driven programming or C++. Although the software designers did have on average around 10 years programming and design experience, the change to an OOD approach represented a paradigm shift which called into question the relevance of the past experience. OOD techniques and representations proved difficult to discover and apply which resulted in the use of adapted conventional representations which team members were familiar with. In spite of the changes in approach, some skills and experience of the technical core of the team were still relevant, e.g. some programmers had an aptitude for learning and using complicated code syntax, which was still a useful skill in C++. However, for Windows programming the programmers had to learn the scope of vast code libraries that were available, e.g. Borland's Object Windows Library (OWL), which involved a significant learning curve.

The software company management recognised that their software designers and programmers were highly skilled but thought that the transition from traditional programming techniques and languages would not present a problem. This is an understandable attitude as it is usual for programmers to be learn new programming languages with relative ease. However, the paradigm shift from traditional to an object-oriented approach needed for the Seed system was not fully appreciated.

The technical core of the team also had no experience of working with an HCI designer, and little experience of working with knowledge engineers. Thus changes in roles and responsibilities of the team was a significant influence on the development.

1.9.2 Competence

There are two ways of looking at the competence of the software team members. The first concerns a team member's adequacy of performance on a specific task, i.e. programming. The second relates to the team member's proficiency of performance in their role within the organisational culture.

Due to inexperience with OOD, C++ and Windows, several members of the technical core of the team were not conversant with the new techniques at the outset of the project. Therefore, by the first definition their competence could be questioned. However, the competence of some individuals could be more easily called into question by considering their competence from the perspective of organisational culture. The software company development culture demanded effective teamwork and placed more emphasis on this aspect than it did on written documentation like the specification.

Therefore, team members that were unable to work in this way could be considered culturally incompetent. There were many examples of this during the Project 1 where cultural expectations of an individual performing a particular role were not realised.

1.9.3 Attitude

The older and more experienced technical core of the team held traditional software engineering values and often did not present a positive attitude to new approaches, new ideas or new roles on the team. For example, it was very clear that the technical core of the team were keen to see a full and comprehensive specification document, from which the work could be divided up into portions which could be tackled by individuals. The new culture which did not support comprehensive specifications and put greater reliance on the skills of the team, was not well received.

There appeared to be a lack of respect between different disciplines in the team, and even within disciplines. The technical core of the team had very little respect for the values of the HCI designer and his user concerns. Conversely, the HCI designer had little respect for the motivations of the technical core, who valued the elegance of the internal code over that of the visual interface. Management attitudes also contributed to attitude problems as programming was often heard to be called, “the easy bit” of the development. Management attitudes towards HCI were generally more flattering.

The job of programming the Seed system actually became somewhat of a thankless task as the HCI designer would be recognised for his design efforts and the programmers were merely expected to build the software precisely to the design. If a programmer wanted to excel, it would have to be by working faster - they were not in a position to improve on the design (only deviate from it) and any coding that was clever would go unnoticed by management.

A further contribution to negative attitudes in the team was the level of fear that seemed prevalent. This could be observed as programmers fearing that they would not be able to produce code to meet the UI design, or experienced technical people fearing new blood, or the fear of appearing foolish in front of peers. Fear appeared to create resentment and negative attitudes and resulted in clear barriers to communication and the free flow of ideas.

1.9.4 Roles and Responsibilities

New roles appeared in the software company’s software teams for the first time in Project 1, so conventional responsibilities of roles in the team changed. For example, the introduction of the HCI designer role caused resentment among some programmers who felt that they had lost a creative outlet as they could no longer design their own UI.

Roles and responsibilities within the development team were deliberately not clearly defined so that new roles, such as HCI design could develop. However, ambiguity of roles and responsibilities often resulted from this flexible approach.

The roles and responsibilities within the team were diverse illustrating the multi-disciplinary nature of the team, as can be seen in table B.9.

Table B.9 Illustration of some of the roles and responsibilities within the Project 1 software development team

Formal Roles	**Responsibilities**
HCI designer	• elicitation of user requirements • ensuring user issues are considered throughout the project • designing the UI • assuring quality of the interaction interface
Software Designer	• ensuring interaction design is feasible • estimating internal design and programming effort required to produce software • producing an internal design of the software (High Level Design - HLD) • representing the HLD in the Software Specification • coordinating the work of the programmers
Programmer	• ensuring that code performs according to specification and HLD • producing low level design • producing sensible code which is efficient, maintainable, re-usable, and unambiguous
Knowledge Engineer	• gain domain understanding, elicit expert knowledge and construct a knowledge model • validate the knowledge model • monitor implementation of KBS, and test the results produced
Project Manager	• ultimate responsibility for delivering software to Time, Budget and Quality (TBQ). • ensuring effective teamwork • producing overall project plan • monitoring progress • reporting to management

The responsibilities associated with the roles of the HCI designer and those of the technical core of the team were often in conflict. HCI design activities would regularly complicate the tasks of the technical core of the team for reasons associated with the UI and the user's perspective. Corners that were cut by the technical core of the team were often unacceptable from a user perspective. The HCI designer's assumed responsibility for the UI to the software necessitated involvement in all aspects of the project, especially working closely with programmers.

Some roles had changing responsibilities throughout the project, for example, the HCI designer was initially responsible for user requirements elicitation, then design of the UI and finally ensuring the quality of the UI by reviewing the work of the programmers. This final responsibility for assuring quality overlapped with the project managers QA responsibilities, causing the HCI designer to appear to be a kind of deputy project manager towards the end of the project.

Within the roles defined in the table B.9 other informal roles were apparent. This was particularly the case for programmers (including software designer/programmers) whose individual differences and characteristics created a different informal role for each of them. Table B.10 illustrates the informal roles of individual programmers on the team. These informal roles often contributed to defining what work each programmer would do. For example, the *Hacker* would often be used to find out if a proposed technical solution was possible, however their unstructured approach often did not give a clear indication of how long it would take to achieve. The *Finisher* was a programmer committed to precision and structure and so was an excellent meticulous GUI programmer.

Table B.10 Informal roles which evolved from individual differences between programmers on the Project 1 team

Informal Programmer[2] Role	**Characteristics of individual**
The Hacker	• strong aptitude with programming syntax • pushes the limits of what the programming language is capable of • has a good understanding of the capabilities of the programming language • creative approach to programming • highly unstructured approach to work • poor programming technique • poor documentation of code • no respect for standards
The Finisher	• fairly good grasp of programming syntax • unimaginative approach to programming • highly structured approach to work • clear and simple programming technique • good documentation of code • precise and tidy approach to GUI coding • good understanding of standards
The Critic	• quick to spot possible exception cases early on in the design phase
The Old Timer	• solid approach to programming based on tried and trusted techniques • fearful of, and resistant to, new techniques and ideas • unimaginative approach to programming • little respect for UI concerns
The Whiz Kid	• good all rounder • imaginative and creative approach • enthusiastic • early 20's • highly skilled and educated in computer science • good documentation of code • commercially inexperienced

Loose definition of roles and responsibilities and the utilisation of individuals informal roles seemed to have great potential in team oriented software development. However, the Seed development team were not a cohesive team oriented unit but rather a collection of individuals working on the same software. In this environment, ambiguity of roles and responsibilities did appear to further hamper communication within the team.

[2] Some of these informal roles relate to individuals who are software designers as well as programmers

1.9.5 Motivation

Motivations of members of the Seed team were often distinct. Members of the technical core of the team were often motivated towards producing simple, neat and elegant code, a motivation which stemmed from traditional software engineering values. The HCI designer's motivation was to produce the most appropriate UI for the users of the Seed, which often caused the internal code to become complex and sometimes untidy. Furthermore, the technical core of the team were motivated towards making the programming task easier for themselves, so resisted complex UI requirements, often with complete disregard for the domain requirements. Fundamentally, the perceptions and motivations of the technical core of the team were attuned to functionality and not interactivity concerns. During the project, few technical people read the HCI 'non-functional requirements' document and this may be attributed to their purest technical motivations which may translate 'non-functional requirements' into 'no need to read this'.

1.9.6 Personality

The personalities of members of the Seed team appeared to have significant impact on the performance of the team. A significant proportion of the technical core of the team were observed to have poor social and communication skills and often preferred to work alone. Examples of this included, non-participation and apparent lack of interest in project meetings, unwillingness to seek clarification on various technical points, preferring to make assumptions and general willingness to act unquestionably on the instructions of others. Younger and more inexperienced team members for example, the HCI designer and the knowledge engineer, out of necessity, took on more responsibility in managing the development, based largely on their social skills. Examples of this included facilitating discussions in meetings with unwilling participants, following progress and reviewing work of technical team members as well as providing and specifying their work. Personalities, skills and performance of members of the technical core of the team in particular, were seen to be diverse.

1.9.7 Conclusion

Various characteristics of individuals within the Seed team was believed to have had considerable influence on the software development. Members of the technical core of the team in particular, appeared to have diverse skills and abilities and generally poor social and communication skills, often preferring to work alone. The impact of some of the characteristic described on communication and collaboration within the team would appear to be of particular importance.

1.10 Changes in requirements

During the course of Project 1 the software requirements regularly changed causing disturbances to the development process. Some of these changes came from influences which were external to the team and some arose from within the team. Regardless of the source or motivations for the changes they had an important impact on the development. Table B.11 summarises considerations, observations and indicators of the causes and effects of changing requirements. The following sections explain the issues in greater depth.

Table B.11 Considerations, observations and indicators of the causes and effects of changing requirements

Considerations	Observations	Project 1 Indicators
External Influences	Requirements not well specified	• Insufficient domain knowledge at the outset of the project • Requirements were ambiguously stated
	Senior level disagreement over project objectives	• Sales and technical areas had been exposed to different factions within the industry and had therefore developed ideas from both ground level technical problems and from top level management concerns.
	Continual exposure to target industry	• Knowledge elicitation visits to ICI. • Sales visits to potential clients • Process industry hard hit by the recession
Internal Influences (within the team)	Some early software concepts proved inappropriate	• Team members unfamiliar with Windows, OOD and C++ • HCI designer was inexperienced
	New ideas were generated as project progressed	• As exposure to the problem increased, team members gained a better understanding of it and so had new ideas.
	Some elements of the code were implemented inconsistent with design intent	• Misinterpretation and misunderstandings coupled with a lack of familiarity with Windows and C++.
	Improved visualisation from examination of elements of the final working system	• When the HCI designer saw previous static designs suddenly animated in a working system, he was in a better position to criticise the UI design. • The inexperience of the HCI designer perhaps contributed to this.
Impact on the development	Changes all came through the HCI designer	• Disrupting the work of the HCI designer • Establishing the HCI designer as the cause of the changes in the eyes of some team members
	The technical core of the team resisted changes	• The technical core failed to regard changes as improving quality. • The development process motivated the technical core to concentrate on the speed of implementation. Changes threatened their chances of delivering an aspect of the implementation within initial estimates.
	Management of the changes	• Agreed with management. • Issue of a specification update. • HCI designer logged UI changes and reasons • Paper-based bug/enhancement request system set up.

1.10.1 External Influences

Changes in requirements of the Seed system were motivated, in part, by influences which were external to the project team. There were three sources of external influence to the Seed project, corresponding to the initial statement of requirements, disagreements over project objectives and continual exposure to the target industry.

The requirements for the Seed were not well specified at the outset of the project due to a recognised lack of knowledge about the Process industry domain. Furthermore, requirements that were specified were often ambiguous, not widely understood in the team and not constraining.

Disagreement at senior level between technical and sales people over the objectives of the Seed project filtered through to the development team as confusion over requirements. The commercial importance of the project further fuelled disagreements at senior level and enhanced the confusion over requirements.

Continuing knowledge elicitation visits to ICI and further sales exposure led to gradually improving domain knowledge throughout the project duration. As domain knowledge improved, earlier specified requirements were called into question and new requirements were added. It should be noted that at this time the Process industry was being hard hit by the recession which caused its outlook and priorities for software to change. The changing priorities of the industry generated new requirements for the software.

1.10.2 Internal Influences

Other demands for changes to the Seed system came from within the project team itself for a variety of reasons.

The innovative and complex nature of the Seed system meant that some of the early software concepts (including UI designs) proved to be inappropriate, thus corrective changes were necessary. These early inappropriate concepts and incorrect decisions were contributed to by the lack of experience of the HCI designer, and the technical core’s unfamiliarity with Windows, OOD and C++.

As team members’ appreciation of the problem evolved during the early months of exposure to it, new ideas were generated and flaws in old ideas realised, further contributing to the need to make changes.

Changes would also be necessary when a misunderstanding or misinterpretation had occurred causing an aspect of the implementation to be inconsistent with the design intent. This was also contributed to by the teams lack of experience with Windows, OOD and C++.

It was sometimes not until elements of the Seed system were implemented that the HCI designer was able to visualise how the UI would look and feel. At this stage, the HCI designer could begin to see further flaws in the UI design or ways of improving it and so would instigate changes.

1.10.3 Impact on the Development

Whatever the cause, changes in requirements usually went first to the HCI designer in order to redesign the UI. This was particularly disruptive for the HCI designer who still had further design work to do. Furthermore, it caused tension within the team as the technical core blamed the HCI designer for the changes.

The technical core of the team were highly resistant to any changes and apparently failed to see any benefits to the quality of the system from making changes. The process itself appears to be partly to blame for this attitude as members of the technical core estimated the time it would take them to implement each aspect of the Seed system at the outset of the project. Any changes made to an aspect of the implementation would jeopardise their chances of reaching these targets. The technical core were further motivated to resist such changes for the technical reason that it was difficult to make significant changes late on in the software process, after the high level design had been completed and programming had commenced.

Management of changes to requirements took several forms. Firstly, significant changes would be agreed by the software company's management, with input from the team. Secondly, changes which contradicted the Specification document would either be dealt with by issuing a new version of the Specification or handing out update information[3]. Thirdly, the HCI designer logged all UI changes and reasons for those changes. Fourthly, a paper-based bug system was set up to monitor bugs and enhancement requests during the latter stages of the project.

1.10.4 Conclusions

Changes to requirements or 'moving the goal posts' was a significant feature of the Project 1 software development. These changes were observed to put a strain on the team and give rise to many misunderstandings throughout the process.

[3] It should be noted that each team member was usually overloaded with paper (memos, updates, designs, etc.) throughout the process

Appendix C Coding and Conceptualisation of Project 1 Data

1.1 Finalised 'Categories of Influence' used for Content Analysis

Category Number	Category Title
1	Roles
2	Skills, ability and competence of individuals within the team were variable
3	The technical core of the team were motivated to resist changes
4	Representation
5	Changes in requirements
6	People from different disciplines interpreting what they see in different ways
7	Visualisation
8	Lifecycle
9	Changing composition of team
10	Management
11	Prototypes
12	Distribution of domain knowledge and understanding of project goals and objectives
13	Specification
14	Implications of commercial requirements
15	HCI became a project bottleneck
16	Emphasis on individual skills
17	Project complexity
18	Effects of team members personalities
19	Emphasis on individual over team approaches
20	Issues arising from team members having not worked together before
21	Interdisciplinary issues
22	HCI criticality in the design phase
23	Ignorance of true / complete HCI role
24	HCI priority

Note that 22-24 were added by peer analysts during triangulation.

1.2 Instructions Given to Triangulating Analysts

Results of an analysis of a long term software development case study have led to the hypothesis that communication and comprehension were key barriers to the performance of the software team.

A 'bottom up' analysis of the content of the case study documentation was undertaken, as is illustrated in figure 1. In other words, 'units of evidence' were extracted from the text and sorted into 'categories of issues'. The categories were then sorted into the underlying 'themes' according to whether they were thought to effect communication, comprehension, both or neither of these issues.

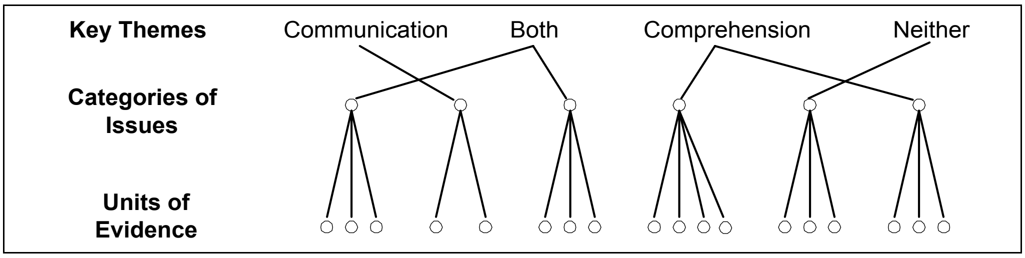


Figure 1. Illustration of bottom-up analysis of data

In order to make the triangulation of analysis manageable, approximately 33% of the 'units of evidence' per category (with a minimum of two units) have been randomly selected. However, all of the categories and themes are still in evidence.

Please perform the following steps in order to carry out the analysis (feel free to ask for clarification and make any comments which you think are appropriate.):

1. Consider each unit of evidence carefully and sort it in to what you consider to be the most appropriate category. If no categories seem to be appropriate, please feel free to create new categories or reject units of evidence if necessary.

2. Sort the categories under into the underlying themes according to whether you perceive that the category relates to Communication, Comprehension, both of these themes or neither of them.

A workshop session will soon be arranged giving the research and triangulating researchers an opportunity to discuss the results of this study. Thanks for your participation in this study.

1.3 Content Analysis Data and Raw Triangulation Results

The ‘information-giving units’ in the following section represent a random selection of one third of the total ‘information-giving units’ sorted into each ‘category of influence’ by the main researcher in the derivation the categories, subject to a minimum of two items per category.

Each ‘unit of evidence’ within the following data set is followed by a three part code. Each part of the code identifies the analyst and the category of influence that the unit was sorted into during the first stage of triangulation. For example, R1, B2,A3 means that the researcher sorted the unit into category one, analyst B sorted it into Category 2 and analyst A sorted it into category 3.

Comments and notes made by the triangulating analysts were recorded and used in the subsequent workshop discussion, they may be found in section 1.4.2.

It should be noted that during the workshop session these raw triangulation results were presented on a large whiteboard with space to show more clearly how the units were sorted into the categories by the researchers.

1.3.1 Information-giving Units and Raw Stage One Triangulation Results

Changes in roles and responsibilities at times created tension and despondency within the team. (R1,B1,A1)

A key part of the investigation phase to me as a new HCI designer, was to try to work out what exactly my role on the project would be. (R1,B1,A1)

Most of the software architects and programmers had considerable skills and experience of computer programming. However, the relevance of the skills and experience is unclear as this technical core of the team were primarily conversant with structured and data driven design and programming, with character-based user interfaces. (R2/B2/A2)

as most constraints... (*of MS Windows*)...were tacit and some programmers were unfamiliar with Windows even as users, inconsistent styles of implementation were common. (R2/B2/A2)

programmer’s (including software architect/programmers) ... individual differences and characteristics created a different informal role for each of them. (R2/B18/A18)

any changes that were required to a module were fiercely resisted by the technical person responsible for it, fearing that the changes would cause them to exceed their initial estimates. In fact, any changes soon became met by a negative reaction (almost a reflex) on the part of some of the technical people. (R3/B3/A3)

The older and more experienced technical core of the team held traditional software engineering values and often did not present a positive attitude to new approaches, new ideas or new roles on the team (R3/B18/A3)

(*The inappropriate nature of DFD representations*) ...was dealt with by adding control flows to the diagram... (*but*) ...this complicated the diagram and was perhaps an inappropriate extension to the representation. (R4,B4,A4)

DFDs... ...proved to be a difficult representation to apply to the architectural design of the Seed system. The primary reason for this is that Windows is an event driven environment and DFDs were designed for data driven environments.(R4/B4/A4)

The main addition that I made... (*to the Demonstrator task model*) ... was to convert the existing hierarchial diagram into a more conventional Hierarchial Task Analysis (HTA)... (*representation*) ...HTA enabled me to make use of the structure of the existing diagrams but also add plans to the diagram to indicate the sequence of tasks, dependencies and to show basic logic... This provided more information to the reader of the diagrams but I still felt this likely to be insufficient for some team members, so I also produced written explanation of the tasks and plans...(R4,B4,A4)

I ... tried to represent the storyboard by arranging all of the sketches on an A2 sheet of paper. There followed a design review which was disjointed, unstructured and unsystematic. The ... representation let the team see the whole picture all at once, allowing people to continually pick up on points that were not relevant to the flow of the walkthrough. The result was that inappropriate details were discussed instead of the overall structure of the visual interface. (R4,B4,A7)

It was felt that the task models should have solved ...(*the problems that team members were having in visualising the Seed system in use*)... but in their existing format did not provide what was required. ... The approach taken to address this problem, was to produce a model of how the engineer's task would be structured if the Seed system were in place, this was called a *Future Task Model* (FTM). I 'invented' this representation ad hoc ... (*as*).... Practical texts, with relevant, applicable and commercially viable techniques or approaches were not available.(R4,B4,A7)

...Windows artifact(*s such as dialog boxes, were captured from other Windows applications, copied into paintbrush and then crudely edited*) ... This approach, which provided a design which looked like a credible Windows artifact, seemed to convey more to the team, especially programmers, than a pencil sketch had done previously. (R4,B4,A7)

The problem of diversely skilled readership was addressed by using different representations within the specification to give different views on the software. (R4,B4,A21)

I ... discovered that the best method of documenting possible solutions involved writing a simple one paragraph description with a visual representation and a pro/con analysis of the option...This approach was geared towards presenting and explaining each option concisely so that a decision could be reached, rather than writing up the options for team members to read for themselves.(R4,B21,A13)

Given the time constraints of the project it was felt that designers should use the techniques and representations that they knew well and adapt them for use with an event driven Windows environment.(R4,B14,A2)

Changes in requirements from the Specification document or from the currently accepted designs proved a considerable disturbance to the development process...(R5,B5,A5)

...the lack of domain knowledge at the outset of the project did not provide a good basis for specifying software requirements or objectives and also led to changes in requirements throughout the project duration (as domain knowledge improved). (R5,B5,A12)

Technical team members appeared to derive information about what data was displayed whilst failing to recognise what may be complex or troublesome user interface mechanisms. (R6,B6,A13)

the effectiveness of accurately conveying the HCI design intent through an animated prototype appeared to depend upon how members of the multi-disciplinary development team interpreted what they saw in the visualisation. (R6,B15,A6)

The FTM proved particularly effective at helping management and sales people and some team members (those with little domain knowledge) visualise exactly how the Seed system would integrate with the user's tasks and how those tasks would be changed.(R7,B7,A7)

As a tool for HCI design, prototyping allowed an experimental approach to design ideas which could be quickly tested and animated in a realistic Windows style of interaction. A benefit of this approach was the aided visualisation of the design that the tool could provide. (R7,B7,A11)

Many aspects of the required interaction, quality and style of the software are demonstrated in the prototype more effectively than in the written specification.(R7,B7,A11)

A major drawback of the waterfall approach adopted was that it did not provide a clear framework for the team to work together within. Ambiguity of roles and responsibilities were not clarified by the lifecycle. (R8,B8,A13)

the representation of the project plan itself emphasised performance of the individual over that of the team, and was linear so that a clear indication of progress could be given to management. (R8,B19,A19)

Resource availability demanded that during the course of the project, the composition of the team and the roles of team members had to change.(R9/B9/A9)

The loss of expertise in the team and addition of new members created a burden of communication on the Specification, which had to on one hand, capture lost expertise and on the other, inform the new team members. The Seed Specification did not perform well in this role due to the poor quality of the document. (R9,B9,A9)

Management also expected software architects and programmers to train themselves on new approaches required by the new tools being used on the Seed. However, the magnitude of transition from structural data driven programming to object-oriented event driven programming (effectively a paradigm shift) was not recognised ...(R10,B10,A10)

...it is usual good software engineering practice to design code elements to be re-usable but on the Seed development some programmers were explicitly told by management not to bother with that.(R10,B10,A10)

Programmers ... appeared to have a great reluctance to look at the prototype... they would usually ask a specific question about how something should be achieved which I would answer using the prototype to demonstrate. (R11,B11,A11)

Unlike the written specification which is often viewed as a form of contract, features shown in the prototype were apparently not considered contractually binding. HCI and quality features implied in the prototype were easily dismissed within the commercially oriented development context.(R11,B14,A11)

The goals and objectives of the Specmaster Seed project were not clear to all members of the development team.(R12,B12,A12)

SfK's culture aimed to encourage its staff to "buy-in" to projects and each actively contribute by gaining some understanding of the problem and the overall solution concept. ...The full benefit of this sentiment was not realised during the project as the software architects and programmers gained little domain knowledge. (R12,B12,A12)

Software architects and programmers rarely sought to gain any domain understanding as they saw this beyond the scope of their role (R12,B1,A1)

It was understood that the Specification could not represent a final reflection of the Specmaster Seed at such an early stage in the development and it was therefore open to amendment. However, the Specification was signed-off at the highest level inside SfK and was used by the project manager as a kind of contract to what had been agreed we would produce. (R13,B13,A13)

in addition to the multi-disciplinary nature of the software team, managers and the pseudo-client/user, each with different backgrounds and perspectives needed to understand the specification at some level.(R13,B13,A14)

The development culture emphasises the team over the written documentation, which should facilitate changes as team members see fit but the development process and the fact that the specification is a form of contract, conspire against the intended culture. (R13,B13,A19)

with a considerable number of unknowns present, initial estimates were crude but the resulting budget was fixed.(R14,B14,A14)

The need to estimate the effort required to construct the Seed was driven by the commercial need to examine the feasibility of and plan the project at the outset. (R14,B14,A14)

one of the reasons for the integration and raised status of the HCI activity was that it was soon seen to be a bottleneck in the process. Ultimately the HCI detailed design activity had up to five programmers awaiting designs before they could progress. The HCI design activity was also late finishing and directly reduced the amount of time available for programming.(R15,B15,A15)

The HCI design activity was on the critical path of the project for a significant period of time during the design phase. (R15,B8,A22)

The information used to produce the Non-functional requirements document came from the picture I had built up of the domain and the proposed solution...(R16,B16,A12)

I... (*wrote*) ...the ***Non-Functional Requirements***... (*document*) ... to highlight the environmental and user considerations that the Seed should address... ...I believed this document to be more of an exercise that I had been set by management to ensure that I had a good grasp of the problem. (R16,B10,A10)

The method of constructing the FTM could be considered craft-based as the model was formed solely from the HCI designer's mental models of the user's current tasks and the proposed Seed system. (R16,B21,A7)

The innovative nature of the Seed also contributed to the inherent complexity of the project as considerable creativity was necessary to construct the software.(R17,B17,A17)

Further complexity was introduced through the selection of the Windows GUI platform, which necessitated the adoption of event driven and object-oriented principles, which proved to be a paradigm shift of approach from structured programming.(R17/B17/A2)

As designers, software architects and programmers had not seen a system similar to the Seed, initial expectations of what would be produced were vague, and mental models had to be largely constructed from scratch. (R17,B6,A7)

The personalities of members of the Seed team appeared to have significant impact on the performance of the team. (R18/B18/A18)

the team oriented approach demanded by SfK to produce the Specmaster Seed did not suit all of ..(*the members of the technical core of the team*)... many of whom would often comment that they were being asked to do something which was, "not in the Spec". (R18/B18/A3)

I attempted to monitor progress that the technical people were making in an informal way in the hope that I catch any misunderstandings early and they could rectified easily. However, several members of the technical core of the team were not open to this approach and would dismiss any guidance or corrections that I made informally, apparently thinking that I was telling them their jobs. (R18/B3/A3)

UI reviews would often very quickly get programmers into a defensive stance, who would then begin to attribute blame for any misunderstandings. (R18/B3/A3)

For implementation, the specification was divided into parts which could be programmed by an individual. (R19,B19,A13)

A final problem with the prototyping approach is that it is possible to spend large amounts of time perfecting a design. At times I isolated myself from the team by becoming too engrossed or too perfectionist with the prototype.(R19,B11,A19)

Only the technical core of the Seed team had worked together before and new roles, skills and diverse disciplinary backgrounds of new team members introduced a certain degree of unfamiliarity. The lack of familiarity brought with it communication difficulties within the now multi-disciplinary team. (R20,B20,A20)

The inexperience of the team of working together affected communication within the team for two key reason. Firstly, because several team members had not worked together before, did not know what to expect from each other and had not attuned to each others abilities. Secondly, because of the tension and resistance to the changing roles and responsibilities of members of the team. (R20,B20,A20)

The software architects saw the HCI designer as a solely a screen designer, and were unhappy about the HCI designer being involved in specifying the structure of the software or anything else which was not directly related to screen layout.(R21,B21,A23)

Programmers would often argue against a UI enhancement by stating that they considered it more important that the system was able to run properly. (R21,B3,A24)

...we (*the programmers and the HCI Designer*)learned to compromise but each party usually felt wronged by compromising their ideals, for me this was the ideal interaction, for programmers the ideal was elegant code. (R21,B6,A21)

1.3.2 Comments made by triangulating analysts

Analyst A

“There are similarities between category 21 & 6”

(21 Interdisciplinary issues
6. People from different disciplines interpreting what they see in different ways)

“*Category 2 refers to technical people. Would be happier to call it ‘evolution of skill base’ or ‘previous knowledge or skill’.”*

(2. Skills, ability and competence of individuals within the team were variable)

“Category 9 is a subset of category 13.”

(9 Changing composition of team
13 Specification)

“Project Complexity (category 17) is ambiguous as it could also refer to the complexities of managing the project.”

(17 Project complexity)

“Categories 12 and 5 are related.”

(12 Distribution of domain knowledge and understanding of project goals and objectives
5 Changes in requirements)

Analyst B

“Categories 18 and 3 are hard to split.”

(18 Effects of team members personalities
3 The technical core of the team were motivated to resist changes)

“Categories 7, 13 and 4 could be included in a lot of other categories”

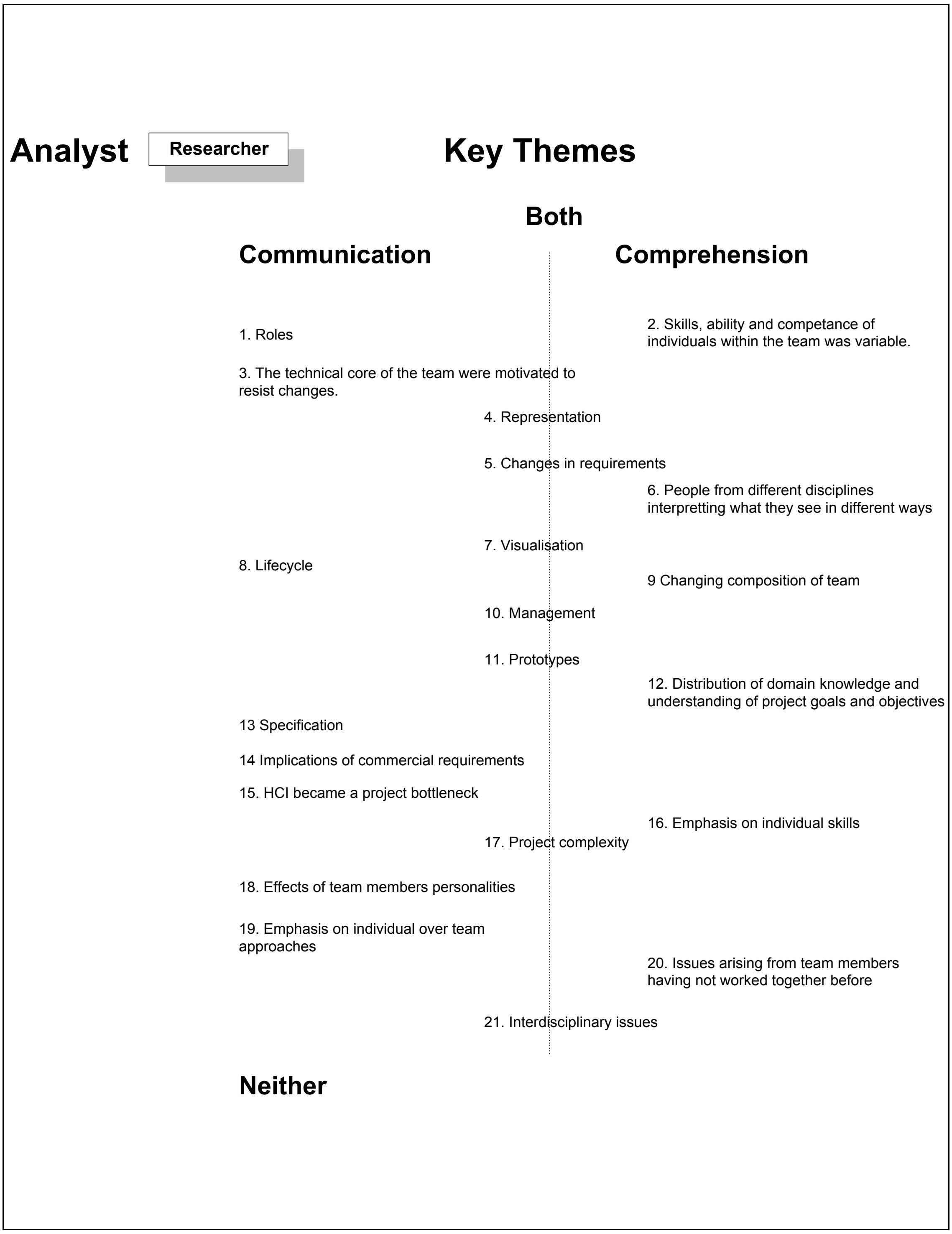
(7 Visualisation
13 Specification
4 Representation)

“Don’t like category 12”

(12 Distribution of domain knowledge and understanding of project goals and objectives)

1.3.3 Raw Stage Two Triangulation Results

Stage Two triangulation involved the analysts sorting the 24 categories according to their relevance to two proposed key themes, ‘Communication’ and ‘Comprehension’. This section contains three figures which depict the raw results.

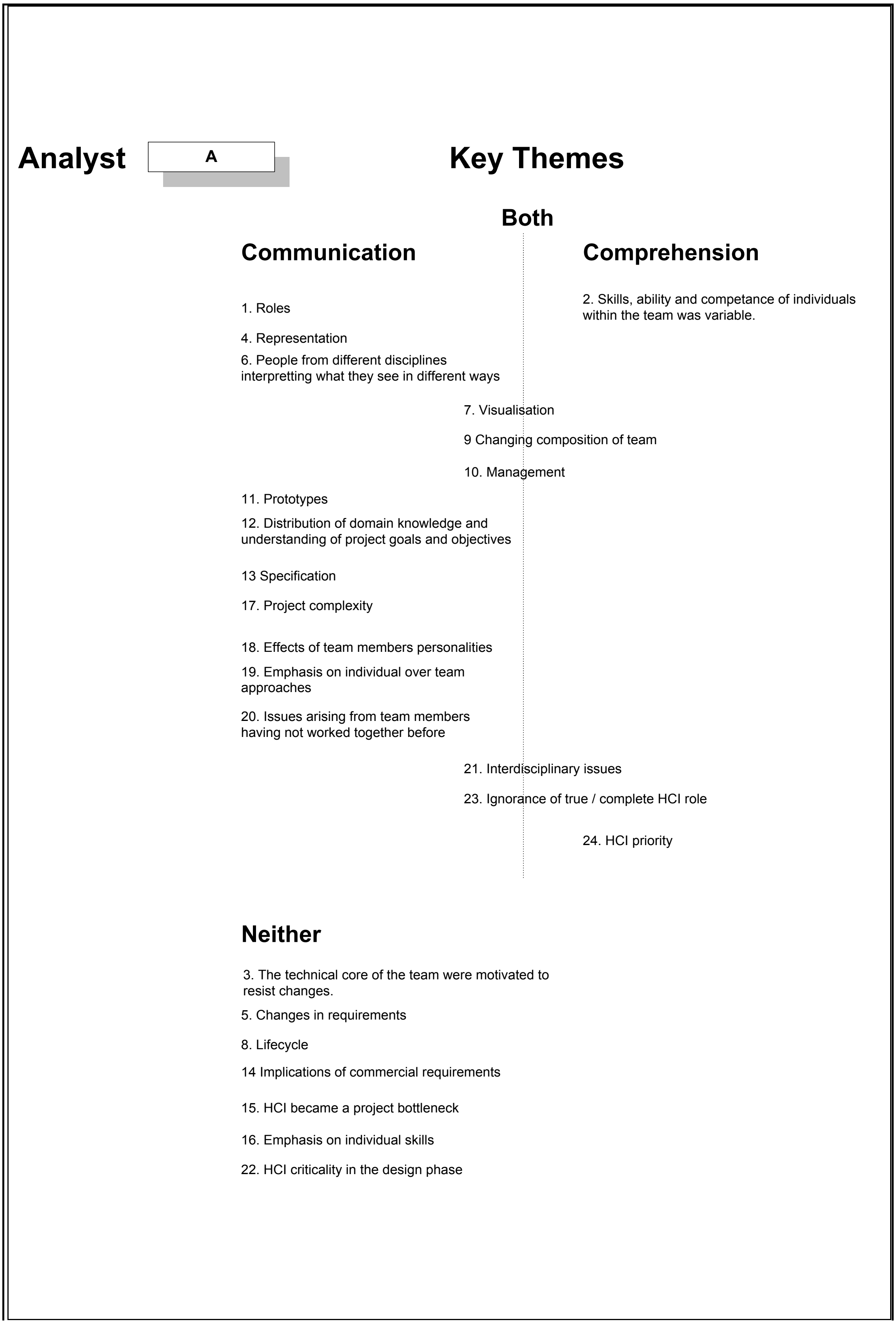


Analyst
Researcher
Key Themes
Both
Communication
Comprehension

1. Roles
2. Skills, ability and competance of individuals within the team was variable.
3. The technical core of the team were motivated to resist changes.
4. Representation
5. Changes in requirements
6. People from different disciplines interpretting what they see in different ways
7. Visualisation
8. Lifecycle
9 Changing composition of team
10. Management
11. Prototypes
12. Distribution of domain knowledge and understanding of project goals and objectives
13 Specification
14 Implications of commercial requirements
15. HCI became a project bottleneck
16. Emphasis on individual skills
17. Project complexity
18. Effects of team members personalities
19. Emphasis on individual over team approaches
20. Issues arising from team members having not worked together before
21. Interdisciplinary issues

Neither

Figure C.1 Stage two triangulation - Categories sorted into themes by the researcher (R)



Analyst A Key Themes

Both

Communication

Comprehension

1. Roles

2. Skills, ability and competance of individuals within the team was variable.

4. Representation

6. People from different disciplines interpretting what they see in different ways

7. Visualisation

9 Changing composition of team

10. Management

11. Prototypes

12. Distribution of domain knowledge and understanding of project goals and objectives

13 Specification

17. Project complexity

18. Effects of team members personalities

19. Emphasis on individual over team approaches

20. Issues arising from team members having not worked together before

21. Interdisciplinary issues

23. Ignorance of true / complete HCI role

24. HCI priority

Neither

3. The technical core of the team were motivated to resist changes.

5. Changes in requirements

8. Lifecycle

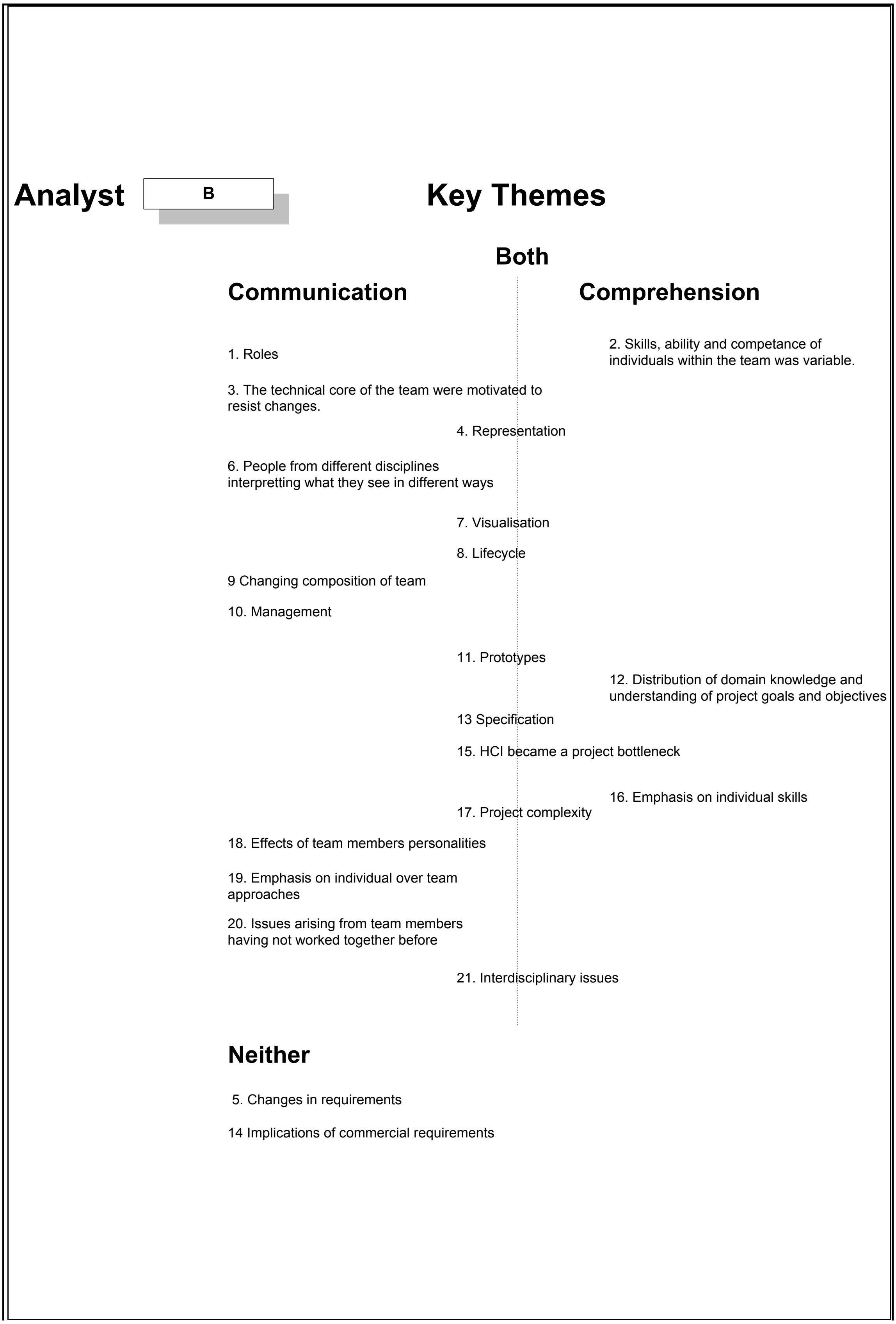
14 Implications of commercial requirements

15. HCI became a project bottleneck

16. Emphasis on individual skills

22. HCI criticality in the design phase

Figure C.2 Stage two triangulation - Categories sorted into themes by Analyst A



Analyst B

Key Themes

Both

Communication

Comprehension

1. Roles

2. Skills, ability and competance of individuals within the team was variable.

3. The technical core of the team were motivated to resist changes.

4. Representation

6. People from different disciplines interpretting what they see in different ways

7. Visualisation

8. Lifecycle

9 Changing composition of team

10. Management

11. Prototypes

12. Distribution of domain knowledge and understanding of project goals and objectives

13 Specification

15. HCI became a project bottleneck

16. Emphasis on individual skills

17. Project complexity

18. Effects of team members personalities

19. Emphasis on individual over team approaches

20. Issues arising from team members having not worked together before

21. Interdisciplinary issues

Neither

5. Changes in requirements

14 Implications of commercial requirements

Figure C.3 Stage two triangulation - Categories sorted into themes by Analyst B

1.4 Discussion Workshop

A workshop session was held to allow the researcher and triangulating analysts the opportunity to discuss the results. Ultimately, workshop activities produced a pseudo hierarchial conceptual model, an accurate representation of which is shown in figure C.6. The model produced has undergone several iterative steps, the end result of which can be found in the main thesis.

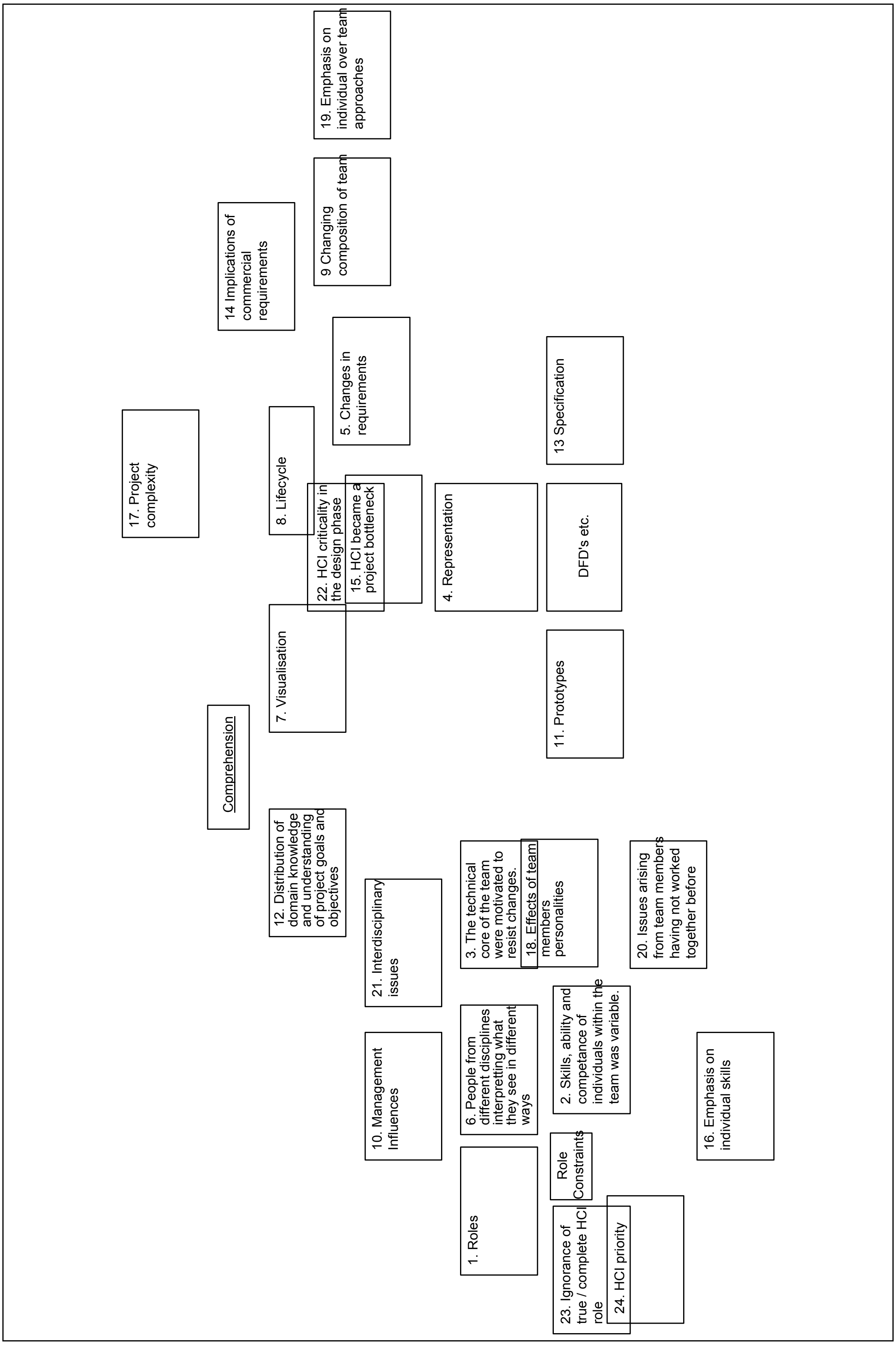

19. Emphasis on individual over team approaches
9 Changing composition of team
14 Implications of commercial requirements
5. Changes in requirements
13 Specification
17. Project complexity
8. Lifecycle
22. HCI criticality in the design phase
15. HCI became a project bottleneck
4. Representation
DFD's etc.
7. Visualisation
11. Prototypes
Comprehension
12. Distribution of domain knowledge and understanding of project goals and objectives
21. Interdisciplinary issues
3. The technical core of the team were motivated to resist changes.
18. Effects of team members personalities
20. Issues arising from team members having not worked together before
10. Management Influences
6. People from different disciplines interpreting what they see in different ways
2. Skills, ability and competance of individuals within the team was variable.
16. Emphasis on individual skills
Role Constraints
1. Roles
23. Ignorance of true / complete HCI role
24. HCI priority

Figure C.4 Accurate representation of Conceptual Model produced during workshop

Appendix D Participant and Expert Assessor Experiment Instructions, 'Model Answers' and 'Marking Frameworks' 450

Appendix D Participant and Expert Assessor Experiment Instructions, ‘Model Answers’ and ‘Marking Frameworks’

1.1 Overview of Participant Instructions

Although the instructions were largely the same for both experimental groups, some differences were necessary for ease of reference as group 1 participants utilised a prototype of Elder while group 2 participants utilised a ProtoTour representation. In this section, parentheses have been utilised to denote differences in instructions given to participants from each group thus, [group 1 instruction variant / group 2 instruction variant]. Comments added to the instructions for the benefit of the reader are shown in italics.

1.2 Instructions to Participants

The exercise you have agreed to participate in forms part of a research project concerned with the use of visual prototypes in the development of complex commercial software.

Answers to questionnaire items and details of your performance in the exercise will be confidential and NOT fed back to your employer. The purpose of the exercise is to assess various aspects of the visual prototyping approach to software development - programmers/designers are NOT being tested.

It will certainly have an adverse effect on the results of this study if some participants begin the exercise with prior knowledge of what the it entails. So, PLEASE DO NOT DISCUSS THE EXERCISE WITH YOUR COLLEAGUES.

The exercise should take no more than an 1½ hours.

Format of the Exercise

The following steps show the format of the exercise:

1. You will be asked to complete a simple questionnaire about yourself and your attitudes towards various aspects of software development.

2. You will receive a 5 minute presentation of a visual prototype which illustrates a proposed software product called ELDER.

3. You will be asked to examine one aspect of the ELDER application with a view to implementation by spending 15 minutes examining [the visual prototype / a representation of the visual prototype (ProtoTour)].

4. You will be asked to spend 45 minutes carrying out several tasks relating to the low level design and implementation of an aspect of the ELDER software utilising the [visual prototype / ProtoTour representation of the visual prototype] (with the support of the interface designer of ELDER who is running this study) as a kind of specification/illustration of the ultimate ELDER application.

5. You will be asked to answer some further questions about the exercise you have just completed.

1.3 Step 1 - Pre-Test Questionnaire

Where you are asked to express an opinion on a 7 point scale please use the full extent of the scale and avoid using middle values in the scale where possible.

Please circle responses where appropriate

Name:

Q1.1 Organisation:

Q1.2 How many years of commercial computer programming/design experience do you have?

<1 1-2 2-5 5-9 >9 years

Q1.3 How much of your computer programming/design experience has involved you implementing the user interface aspect of the software development?

<1 1-2 2-5 5-9 >9 years

Q1.4 Do you have any experience of working on a commercial software development where user-centred design issues had an impact on the way the software was designed (or in other words, a project where development was driven by requirements of the user interface) ?

Yes No

Q1.5 Do you have any experience of working on a commercial software development where a visual prototype (i.e. a prototype showing apparent functionality but no actual functionality, sometimes called a mock-up) was used as part of the specification of the software under development?

Yes No

Q1.6 As a user, how would you describe your familiarity with the Microsoft Windows Help application? (please circle appropriate response)

Completely unfamiliar (never used it)	1	2	3	4	5	6	7	Completely Familiar (regular user)

Even if you have no experience of using visual prototypes, User-Centred Design or User Interface Designers please express an opinion on the following questions...

Q1.7 How do you rate the use of visual prototypes (or mock-ups) within the commercial software development process to aid the design and implementation?

Not Useful At All	1	2	3	4	5	6	7	Extremely Useful

Q1.8 In general, compared with Functionality, how important do you consider User-Centred Design (or user interface issues) to be in the development of commercial software?

Functionality is much more important than User-Centred Design (or the UI)	1	2	3	4	5	6	7	User-Centred Design (or the UI) is much more important than Functionality

Q1.9 How important to the success of a software project do you consider the role of Human-Computer Interaction Designer (or user-interface designer) in a commercial software development team?

Unimportant to the success of the project	1	2	3	4	5	6	7	Important to the success of the project

1.4 Step 2 - Script for Presentation of ELDER Visual Prototype

The following script was read verbatim to each participant and the 'Summary of aspects of the ELDER prototype to present' was used as a guide by the experimenter to ensure that each participant received the same presentation of the ELDER prototype.

A visual prototype (or mock-up) has been constructed for a hypertext engineering advice system called ELDER. The prototype has been developed by a user interface designer and a number of users. The latest version of the prototype (used in this exercise) is felt to be a fairly accurate representation of the ultimate ELDER application.

Following a 5 minute presentation of the ELDER prototype you will be provided with [the visual prototype / a ProtoTour representation of the prototype], from which you will be asked to begin to design a part of the ELDER application and consider implementation issues.

Introduction to ELDER

ELDER is a software tool which aims to provide Engineers with intelligent advice on equipment selection during the plant design process.

The primary goal of ELDER is to reduce the potential hazards in a plant design earlier on in the design phase, thus reducing the need for costly re-work at a later stage.

The Engineer is to be seen as a human expert who must always have full control over the software and full responsibility for the decisions taken with its support. Therefore, the program should be seen as having and advisory role, reacting only to the queries posed by the user.

ELDER Users

There are 4 categories of users of ELDER. The Engineer (responsible for process plant equipment specification), the Supervisor (oversees engineer's equipment specifications), the Author (responsible for the advice and calculations used in ELDER, likely to create and edit advice and calculations) and the System Administrator (responsible for backups, user logins, etc.)

Summary of aspects of the ELDER prototype to present

Main Hypertext Information Area (HIA) - hypertext; diagrams; calculations; split into chapters; search; browser; annotation.

Checklist Window, Log File, Advice and Calculation editor.

1.5 Step 3 - Preliminary Design of ELDER

Tasks which you will be asked to perform in this section relate to the design of one aspect of the ELDER software and other implementation considerations.

The tasks should be performed by utilising the [ELDER visual prototype / ProtoTour illustration of the ELDER visual prototype] **AND with the aid of the HCI Designer (user interface designer) running the exercise**, who will readily supply further information about any aspects of the ELDER software which you feel requires clarification or further explanation.

Complete the following tasks as if you had been given the job of implementing the **Checklist Window** in ELDER. Read all questions carefully before starting the exercise because the questions are closely related and you may wish to answer them in a different order or answer several at a time.

IMPORTANT NOTE

Spend the first 15 minutes investigating the functionality and interactivity of the Checklist Window using the [visual prototype / ProtoTour representation of ELDER] provided (you may make notes if you want to).

In the remaining 45 minutes carry out the following tasks as fully as possible - please try to attempt all tasks if possible.

Note that answer sheets were provided to all participants summarising each task and providing reminders of all aspects of the task requiring completion, e.g. for Q3.5 participants were reminded to note down their estimate as well as the implementation language they estimating for.

Q3.1 Imagine you have been given the job of implementing the Checklist Window. Although in practice you may not need to do much low level design work or what you do may be just included in a code header, in this case please try to express the low level design (LLD) using flow charts, pseudo code, State Transition Diagrams (STDs) or similar representations which would show some of the detailed logic underlying the implementation. Also supply any relevant supporting notes.

Q3.2 List and briefly explain any assumptions you have made in your design.

Q3.3 List aspects of the Checklist Window implementation which you think may present difficulties to the implementation?

Assess the severity of **each** the implementation difficulty identified using a 7 point scale, where ‘1’ represents ‘very minor difficulty’ and ‘7’ represents ‘very severe difficulty’.

Q3.4 Suggest some alternative ways of implementing aspects of the Checklist Window which could speed up the implementation effort or improve the Checklist in other ways.

For **each** alternative you suggest assess the impact it will have on the overall usability of ELDER on a **9 point scale**, where '1' represents 'Significant Reduction in Usability' and 9 represents 'Significant Improvement in Usability'.

Q3.5 Assume you can choose which development language and tools you can use to implement the Checklist Window (in Microsoft Windows) make a very ROUGH estimate of how long (in man days) you think it would take you to implement the Checklist Window.

(in making your estimate you may assume that implementation alternatives you suggested which you rated 3 or higher than are acceptable)

State which programming language and tools your estimate applies to.

1.6 Step 3 - Observation Data

Questions asked by participants whilst carrying out tasks in step 3 were recorded by the experimenter in tables of the following format.

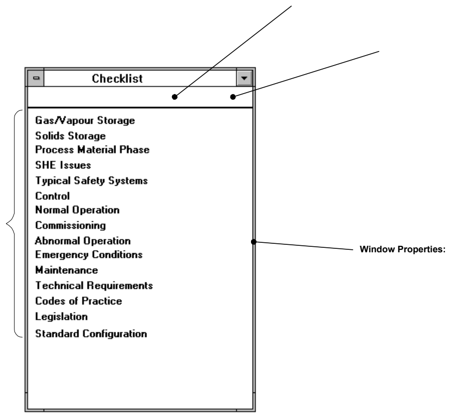
Table D.1 Example of tables used to record frequency and categories of questions asked by participants during completion of tasks in step 3.

Category of question asked by participant	**Question occurrences**
Obs. 3(i) Usability of Prototype / ProtoTour	
Obs. 3(ii) Explanation of Intended Functionality (Design Intent)	
Obs. 3(iii) Design Rationale	
Obs. 3(iv) Users / Tasks	
Obs. 3(v) Scope of the Design Task Set	

1.7 Step 4 Post-test Questionnaire

Q4.1 By thorough annotation of the following picture describe the functionality of the Checklist Window (only a few annotation arrows are shown in the picture but more will be required).

For example, you should define the content and associated functionality of the checklist columns.



Q4.2 Briefly, what is the relationship between the Checklist Window and the Log File?

Q4.3 Briefly, what is the relationship between the Checklist Sections and the Browser Index?

Q4.4 Briefly, what happens to the Checklist Window if the user accesses advice from a Chapter which is different from the current equipment type, e.g. if a Storage Tank is the current equipment type and the user accessed advice from the Pumps Chapter what would happen to the Checklist?

Q4.5 How easy was it for you to visualise how the Checklist Window should work?

Very Hard to Visualise	1	2	3	4	5	6	7	Very Easy to Visualise

Q4.6 How useful were the following aspects of the exercise in helping you to visualise how the ELDER application should work?

(I) Supplemental information provided	Not Useful At All	1	2	3	4	5	6	7	Extremely Useful
(ii) [Your own use of the Prototype during the exercise / Automatic animated walkthroughs]	Not Useful At All	1	2	3	4	5	6	7	Extremely Useful
(iii) The User Interface designer present in the exercise	Not Useful At All	1	2	3	4	5	6	7	Extremely Useful

Q4.7 Do you feel that you were able to get information about WHY the Checklist Window was designed as it is shown in the prototype?

Yes No (If 'No' please skip questions 5. and 6.)

Q4.8 How useful was the rationale for how the Checklist Window was designed in helping to identify implementation alternatives?

Not Useful At All	1	2	3	4	5	6	7	Extremely Useful

Q4.9 How useful was the rationale for how the Checklist Window was designed in helping to assess the effect of implementation alternatives on usability?

Not Useful At All	1	2	3	4	5	6	7	Extremely Useful

Q4.10 How do you rate the use of visual prototypes (or mock-ups) within the commercial software development process to aid the design and implementation [*group 1 sentence ended* / i.e. if they were presented in ProtoTour]?

Not Useful At All	1	2	3	4	5	6	7	Extremely Useful

Q4.11 Supposing that the task you have just undertaken were part of a real software development process, where the HCI Designer (or user interface designer) had used the [Visual Prototype / ProtoTour illustration of the Visual Prototype] to 'hand-over' the outward design of the software to you. Do you have any comments or criticisms about using [visual prototypes / the ProtoTour illustration of the visual prototype] in this way?

Comments and Criticisms...

Q4.12 Overall, would you like to see prototypes used in a similar way to that which has been illustrated in this exercise in future software development?

Yes No

1.8 Overview of Assessor Instructions

Assessor instructions consisted of a brief overview of what the assessor would be participating in and what they would have to do, followed by a pre-defined framework for ‘marking’ the participants’ ‘scripts’. For question 3.1 thorough ‘model answers’ were provided showing detailed design considerations that would need to be made during the implementation of the Checklist Window.

1.9 Assessor Instructions

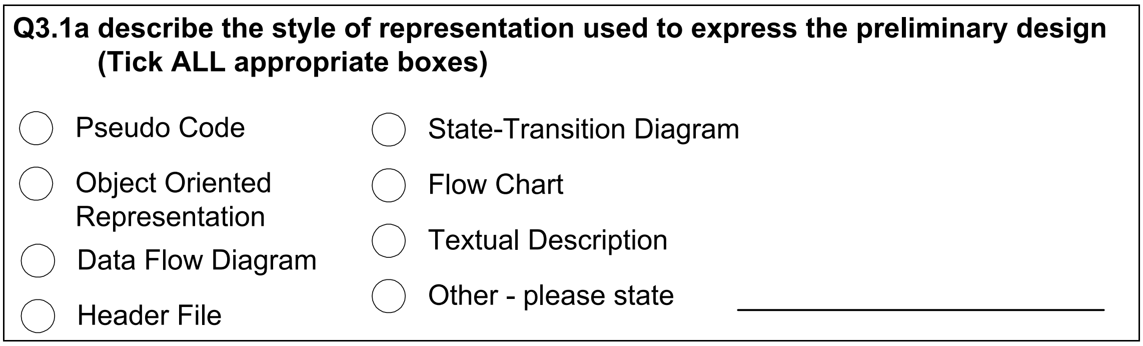
The exercise you have agreed to participate in forms part of a research project concerned with the use of visual prototypes in the development of complex commercial software.

Participants of this exercise have completed various tasks relating to the design and implementation of an aspect of a proposed software application called ELDER. Participants were given different representations of the ELDER application from which to begin to design part of it. You will be asked to analyse and assess responses of all participants but will not be told which representation they used.

Before you can carry out this assessment, you will be required to take part in the exercise as a participant in order to gain an appreciation of what it involves.

After doing the exercise you will be asked to evaluate the answer sheets of all other participants based on the following assessment criteria.

1.10 Assessment of Preliminary Design (Q3.1)



Q3.1a describe the style of representation used to express the preliminary design (Tick ALL appropriate boxes)

○ Pseudo Code

○ Object Oriented Representation

○ Data Flow Diagram

○ Header File

○ State-Transition Diagram

○ Flow Chart

○ Textual Description

○ Other - please state ______________________

Figure D.1 Q3.1a Assessment of the style of representation used for preliminary design

Q3.1b How comprehensive is the participant's preliminary design?

In other words, has the participant considered everything within their design? Or put another way, how much functionality is missing or badly flawed?

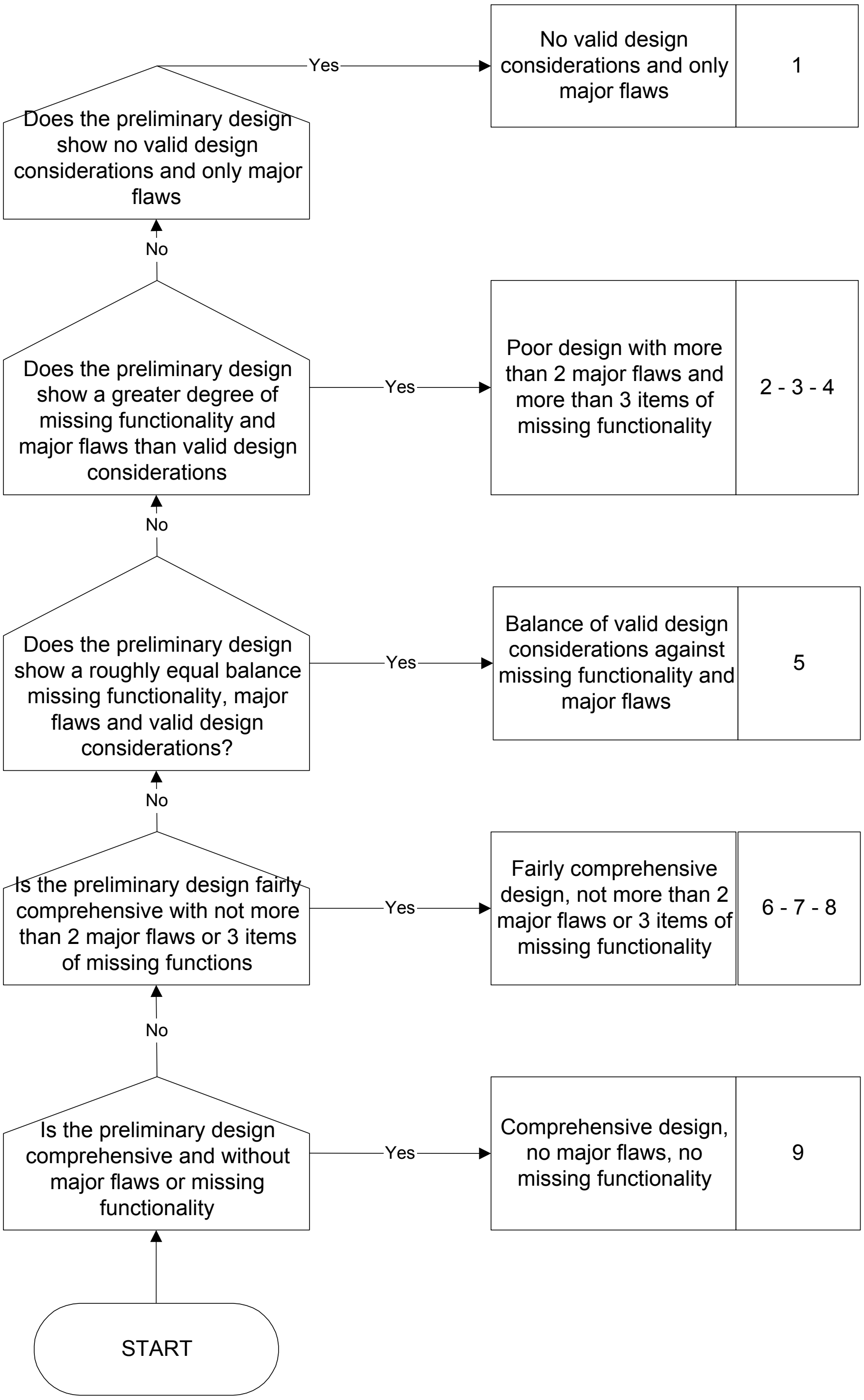

Yes
No valid design considerations and only major flaws
1
Does the preliminary design show no valid design considerations and only major flaws
No
Does the preliminary design show a greater degree of missing functionality and major flaws than valid design considerations
Yes
Poor design with more than 2 major flaws and more than 3 items of missing functionality
2 - 3 - 4
No
Does the preliminary design show a roughly equal balance missing functionality, major flaws and valid design considerations?
Yes
Balance of valid design considerations against missing functionality and major flaws
5
No
Is the preliminary design fairly comprehensive with not more than 2 major flaws or 3 items of missing functions
Yes
Fairly comprehensive design, not more than 2 major flaws or 3 items of missing functionality
6 - 7 - 8
No
Is the preliminary design comprehensive and without major flaws or missing functionality
Yes
Comprehensive design, no major flaws, no missing functionality
9
START

Figure D.2 Q3.1b Assessment of the comprehensiveness of the preliminary design

1.10.1 ‘Model Answers’ to Preliminary Design (Q3.1)

A set of ‘model answers’ was produced to indicate to the assessor the kind of design considerations that surround the implementation of the Checklist Window.

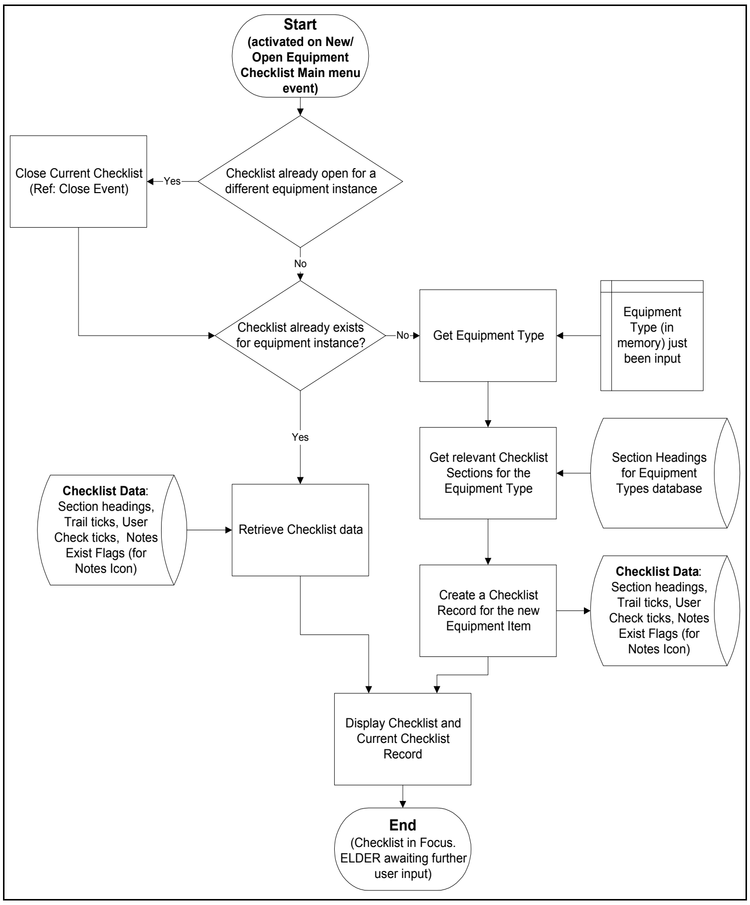

Start
(activated on New/ Open Equipment Checklist Main menu event)
Checklist already open for a different equipment instance
Yes
Close Current Checklist (Ref: Close Event)
No
Checklist already exists for equipment instance?
No
Get Equipment Type
Equipment Type (in memory) just been input
Yes
Get relevant Checklist Sections for the Equipment Type
Section Headings for Equipment Types database
Checklist Data: Section headings, Trail ticks, User Check ticks, Notes Exist Flags (for Notes Icon)
Retrieve Checklist data
Create a Checklist Record for the new Equipment Item
Checklist Data: Section headings, Trail ticks, User Check ticks, Notes Exist Flags (for Notes Icon)
Display Checklist and Current Checklist Record
End
(Checklist in Focus. ELDER awaiting further user input)

Figure D.3 ‘Model Answer’ - Flowchart illustrating required interaction following a New/Open Checklist event

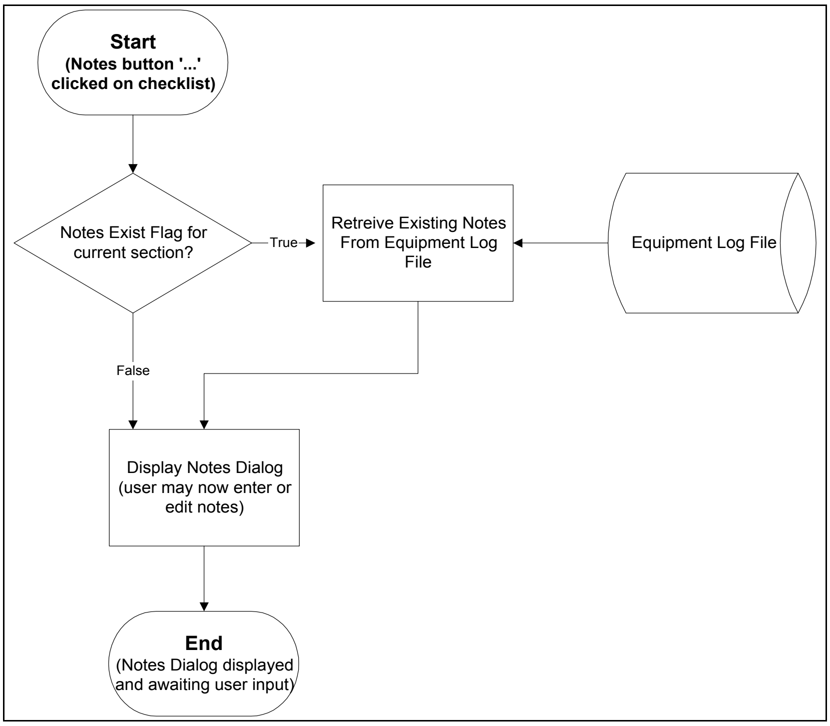

Start
(Notes button '...' clicked on checklist)
Notes Exist Flag for current section?
True
Retreive Existing Notes From Equipment Log File
Equipment Log File
False
Display Notes Dialog (user may now enter or edit notes)
End
(Notes Dialog displayed and awaiting user input)

Figure D.4 ‘Model Answer’ - Flowchart illustrating required interaction following a Notes Button Click Event

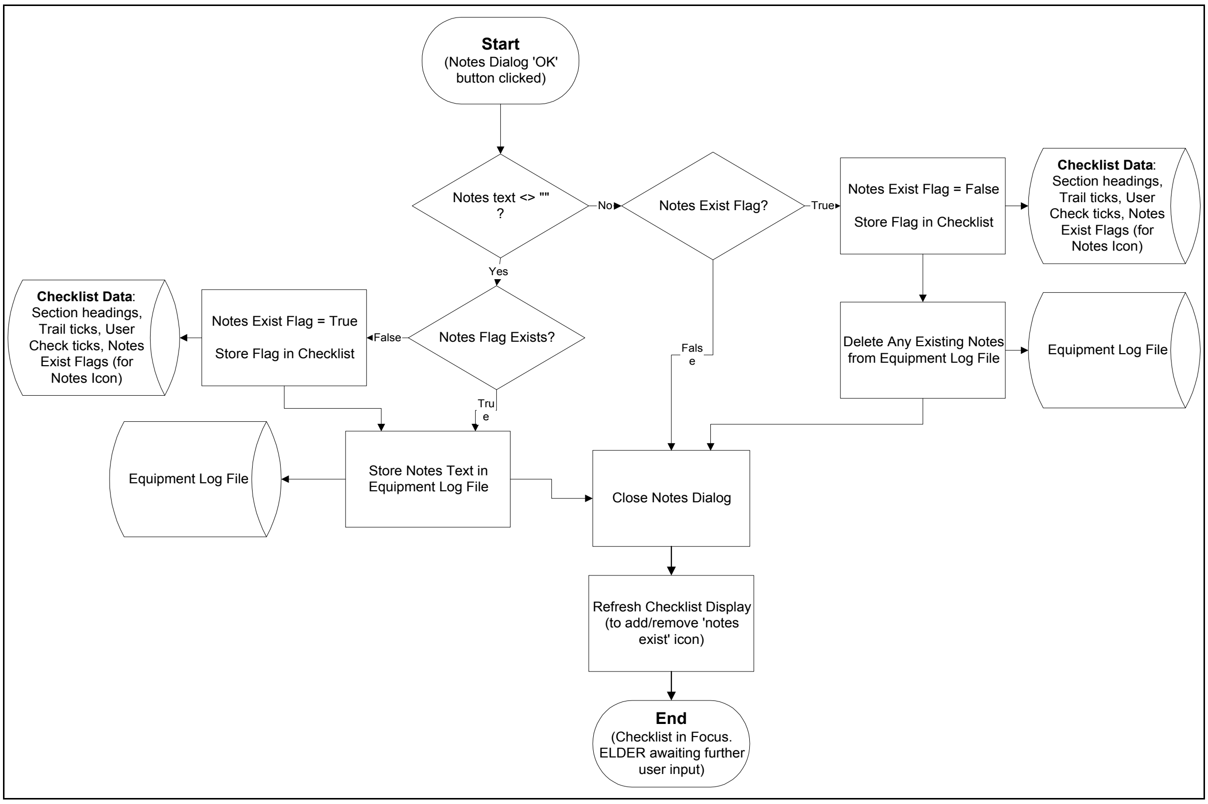

Start
(Notes Dialog 'OK' button clicked)
Notes text <> "" ?
No
Notes Exist Flag?
True
Notes Exist Flag = False
Store Flag in Checklist
Checklist Data: Section headings, Trail ticks, User Check ticks, Notes Exist Flags (for Notes Icon)
Yes
Notes Flag Exists?
False
Notes Exist Flag = True
Store Flag in Checklist
Checklist Data: Section headings, Trail ticks, User Check ticks, Notes Exist Flags (for Notes Icon)
Fals e
Delete Any Existing Notes from Equipment Log File
Equipment Log File
Tru e
Equipment Log File
Store Notes Text in Equipment Log File
Close Notes Dialog
Refresh Checklist Display (to add/remove 'notes exist' icon)
End
(Checklist in Focus. ELDER awaiting further user input)

Figure D.5 ‘Model Answer’ - Flowchart illustrating required interaction following an OK Button Click Event within the Notes Dialog

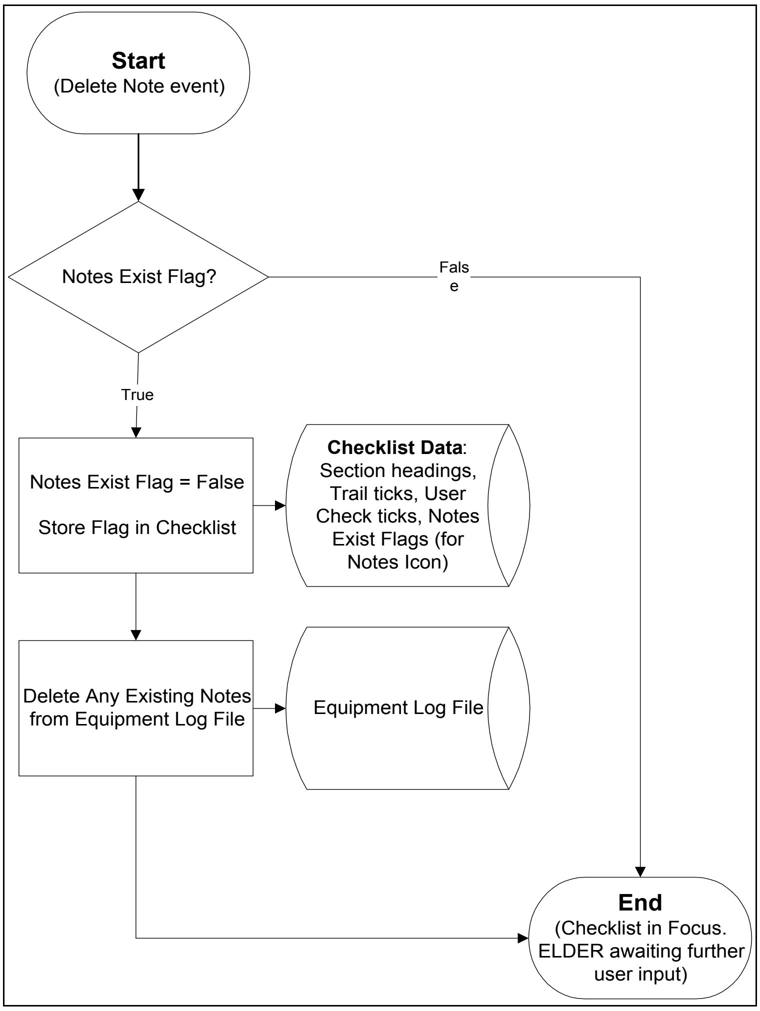

Start
(Delete Note event)
Notes Exist Flag?
Fals
e
True
Notes Exist Flag = False
Store Flag in Checklist
Checklist Data: Section headings, Trail ticks, User Check ticks, Notes Exist Flags (for Notes Icon)
Delete Any Existing Notes from Equipment Log File
Equipment Log File
End
(Checklist in Focus. ELDER awaiting further user input)

Figure D.6 'Model Answer' - Flowchart illustrating required interaction following a Delete Note Event

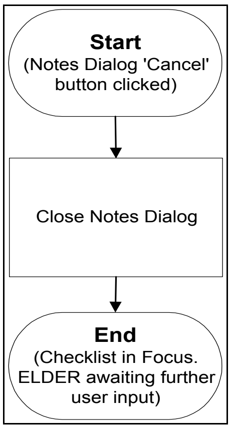

Start
(Notes Dialog 'Cancel' button clicked)
Close Notes Dialog
End
(Checklist in Focus. ELDER awaiting further user input)

Figure D.7 'Model Answer' - Flowchart illustrating required interaction following a Cancel Button Click Event within the Notes Dialog

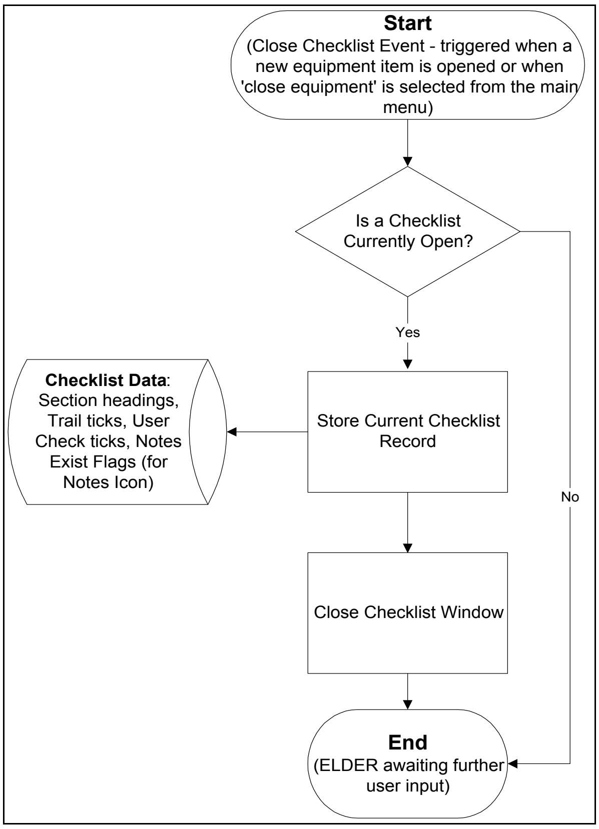

Start
(Close Checklist Event - triggered when a new equipment item is opened or when 'close equipment' is selected from the main menu)
Is a Checklist Currently Open?
Yes
No
Checklist Data: Section headings, Trail ticks, User Check ticks, Notes Exist Flags (for Notes Icon)
Store Current Checklist Record
Close Checklist Window
End
(ELDER awaiting further user input)

Figure D.8 ‘Model Answer’ - Flowchart illustrating required interaction following a Close Checklist Event

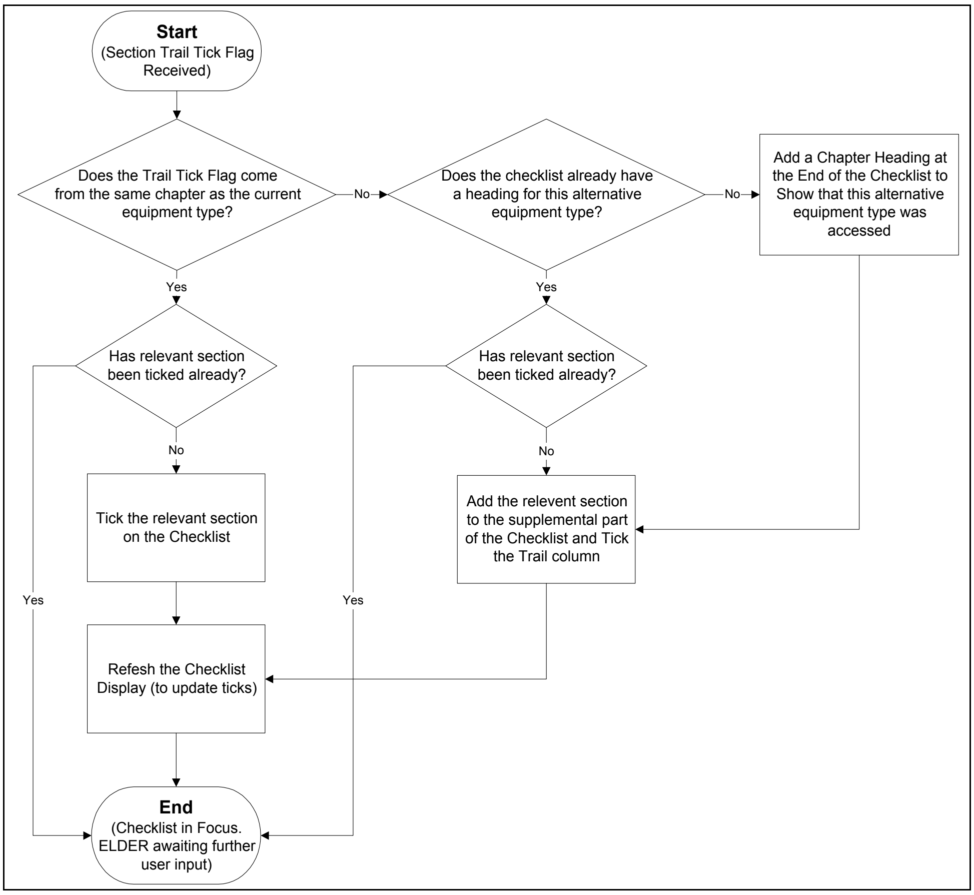
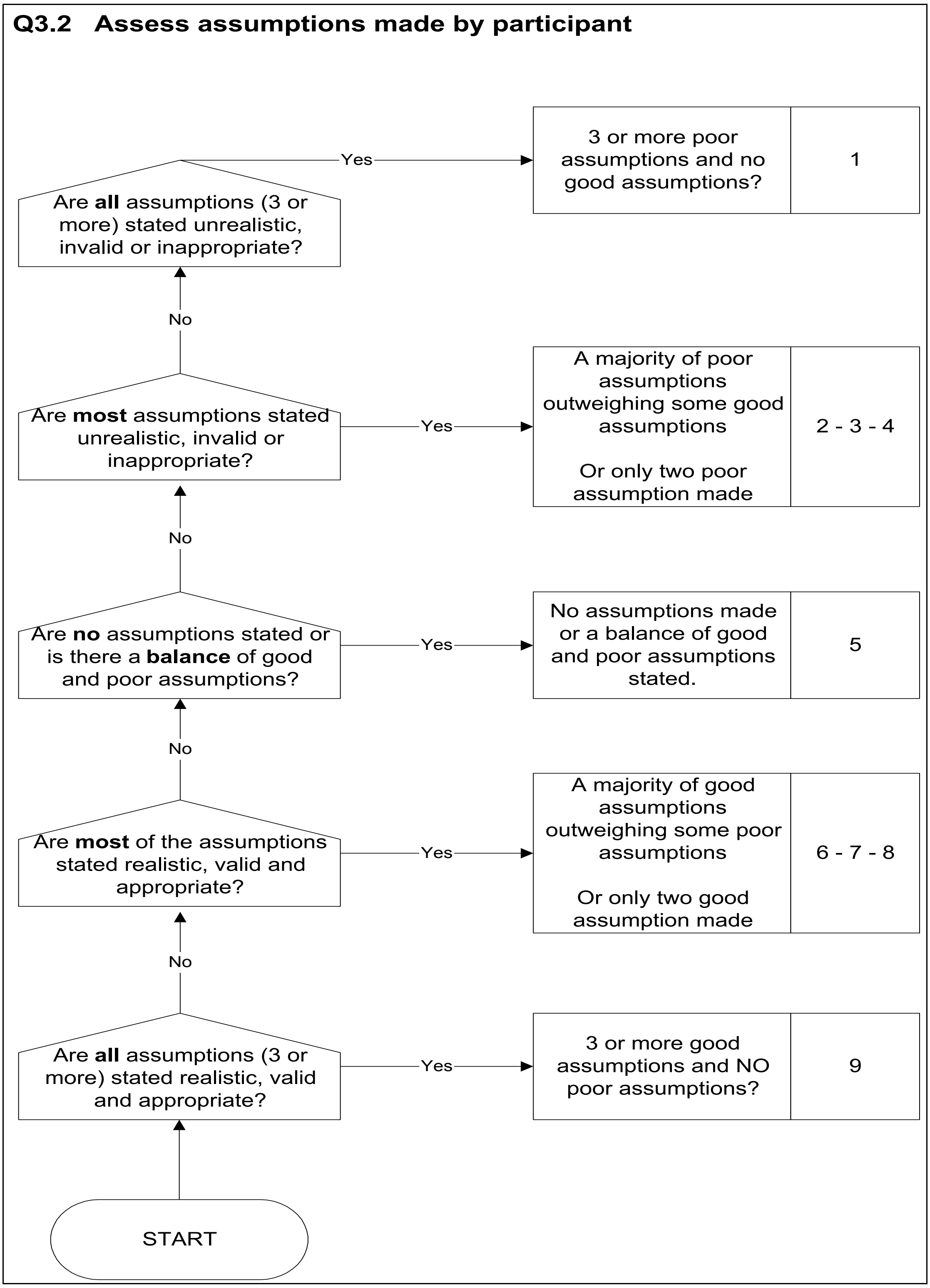

Start
(Section Trail Tick Flag Received)
Does the Trail Tick Flag come from the same chapter as the current equipment type?
No
Does the checklist already have a heading for this alternative equipment type?
No
Add a Chapter Heading at the End of the Checklist to Show that this alternative equipment type was accessed
Yes
Yes
Has relevant section been ticked already?
Has relevant section been ticked already?
No
No
Tick the relevant section on the Checklist
Add the relevent section to the supplemental part of the Checklist and Tick the Trail column
Yes
Yes
Refesh the Checklist Display (to update ticks)
End
(Checklist in Focus. ELDER awaiting further user input)

Figure D.9 ‘Model Answer’ - Flowchart illustrating required interaction following a the receipt of a Section Trail Tick Message

1.11 Assessment of Assumptions (Q3.2)


Q3.2 Assess assumptions made by participant
Yes
3 or more poor assumptions and no good assumptions?
1
Are **all** assumptions (3 or more) stated unrealistic, invalid or inappropriate?
No
A majority of poor assumptions outweighing some good assumptions

Or only two poor assumption made
2 - 3 - 4
Are **most** assumptions stated unrealistic, invalid or inappropriate?
Yes
No
No assumptions made or a balance of good and poor assumptions stated.
5
Are **no** assumptions stated or is there a **balance** of good and poor assumptions?
Yes
No
A majority of good assumptions outweighing some poor assumptions

Or only two good assumption made
6 - 7 - 8
Are **most** of the assumptions stated realistic, valid and appropriate?
Yes
No
3 or more good assumptions and NO poor assumptions?
9
Are **all** assumptions (3 or more) stated realistic, valid and appropriate?
Yes
START

Figure D.10 Q3.2 Assessment of assumptions stated by participants

1.12 Assessment of Implementation Difficulties (Q3.3)

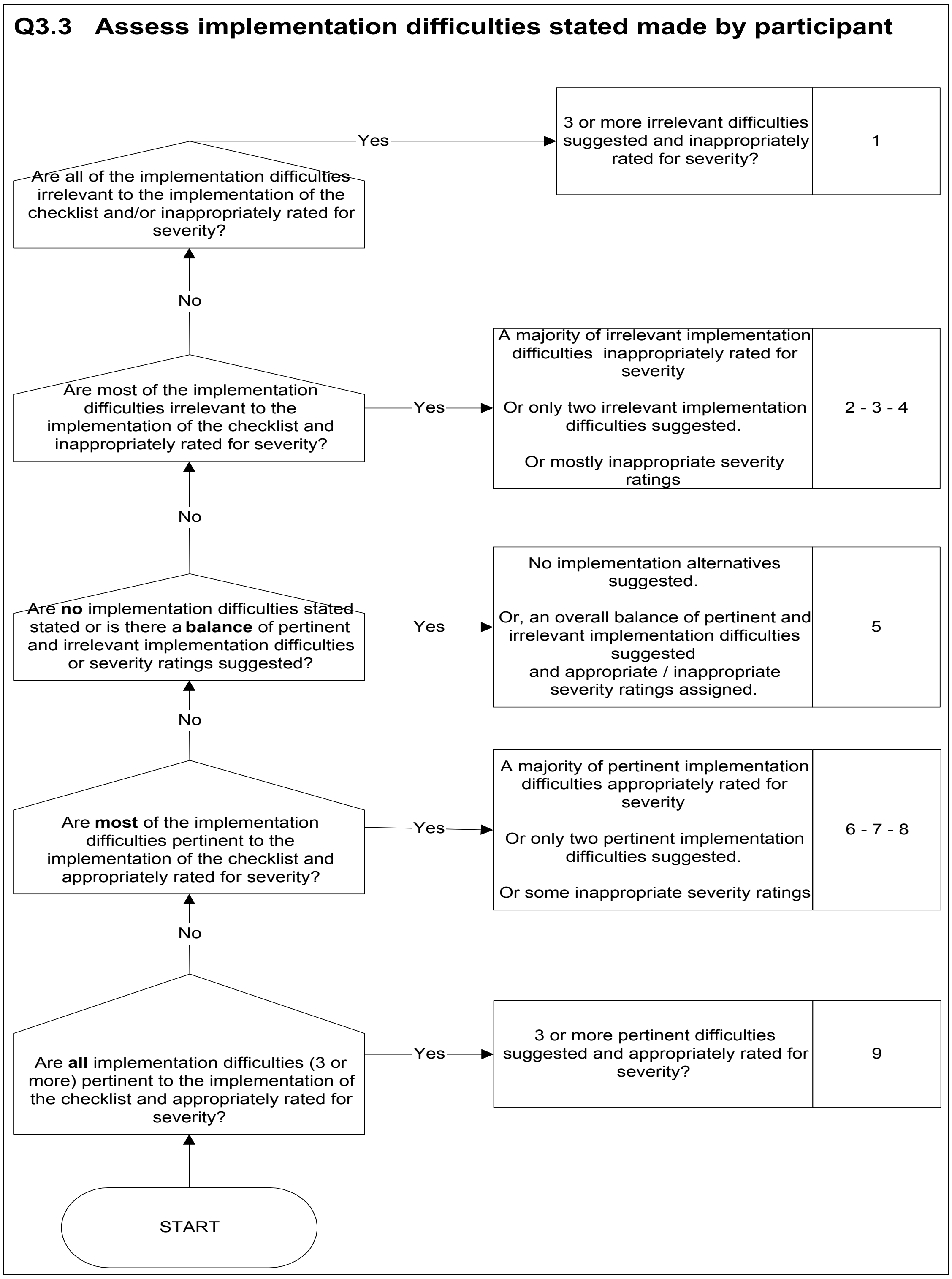

Q3.3 Assess implementation difficulties stated made by participant
Are all of the implementation difficulties irrelevant to the implementation of the checklist and/or inappropriately rated for severity?
Yes
3 or more irrelevant difficulties suggested and inappropriately rated for severity?
1
No
Are most of the implementation difficulties irrelevant to the implementation of the checklist and inappropriately rated for severity?
Yes
A majority of irrelevant implementation difficulties inappropriately rated for severity
Or only two irrelevant implementation difficulties suggested.
Or mostly inappropriate severity ratings
2 - 3 - 4
No
Are no implementation difficulties stated stated or is there a balance of pertinent and irrelevant implementation difficulties or severity ratings suggested?
Yes
No implementation alternatives suggested.
Or, an overall balance of pertinent and irrelevant implementation difficulties suggested and appropriate / inappropriate severity ratings assigned.
5
No
Are most of the implementation difficulties pertinent to the implementation of the checklist and appropriately rated for severity?
Yes
A majority of pertinent implementation difficulties appropriately rated for severity
Or only two pertinent implementation difficulties suggested.
Or some inappropriate severity ratings
6 - 7 - 8
No
Are all implementation difficulties (3 or more) pertinent to the implementation of the checklist and appropriately rated for severity?
Yes
3 or more pertinent difficulties suggested and appropriately rated for severity?
9
START

Figure D.11 Q3.3 Assessment of potential implementation difficulties stated by participants

1.13 Assessment of Implementation Alternatives (Q3.4a)

3.4a Assess implementation alternatives suggested by participant

When assessing alternatives greater credit should be given to suggestions which genuinely relate to the users and their tasks, than suggestions which are aimed at improving the general look and feel of the application for the sake of it (i.e. without sensible design rationale).

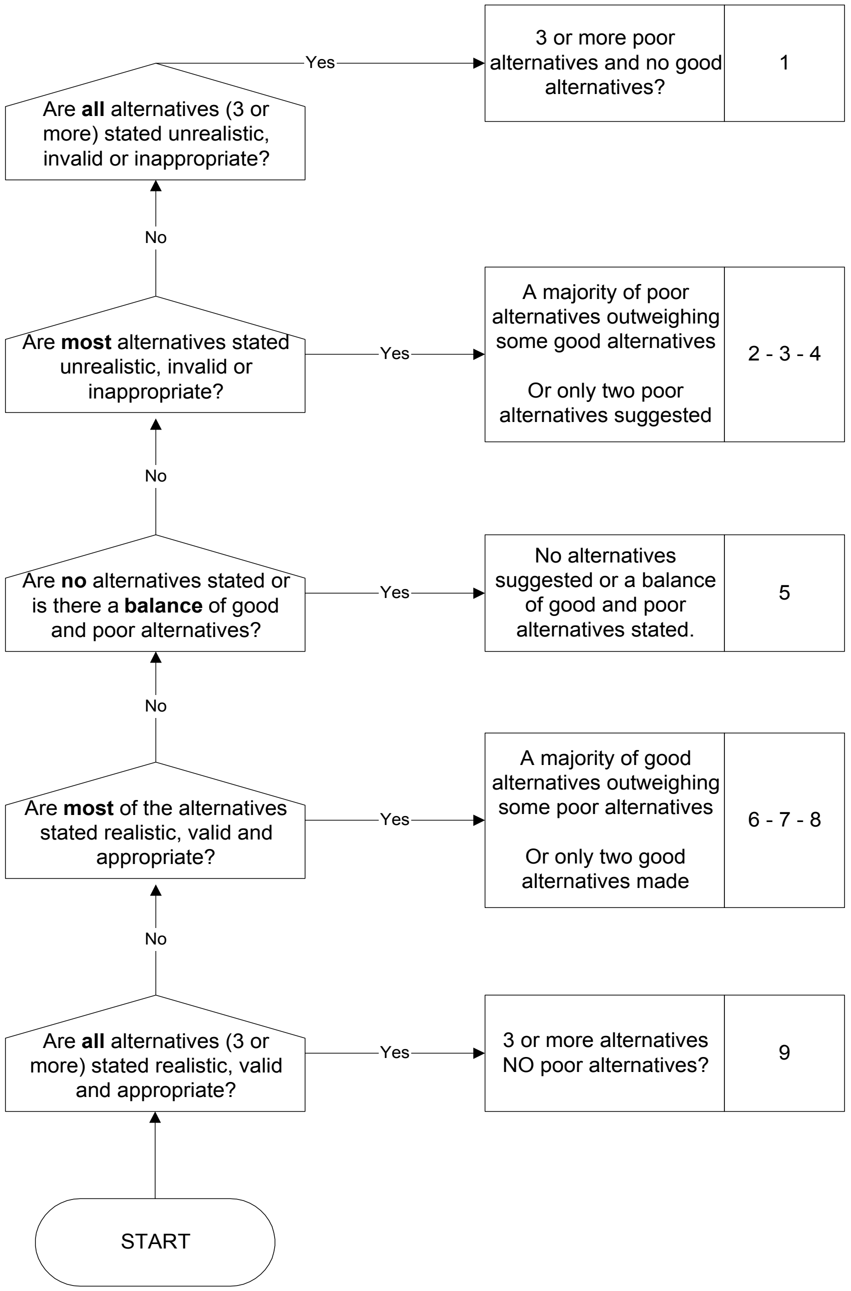
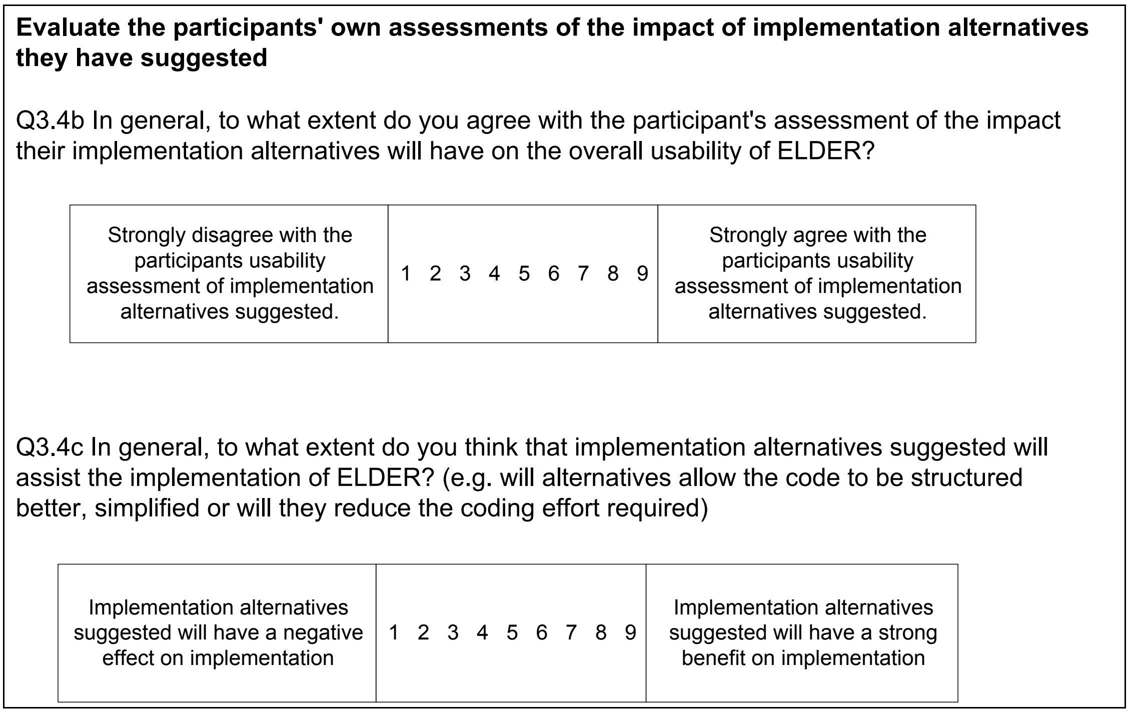

START
Are **all** alternatives (3 or more) stated realistic, valid and appropriate?
Yes
3 or more alternatives NO poor alternatives?
9
No
Are **most** of the alternatives stated realistic, valid and appropriate?
Yes
A majority of good alternatives outweighing some poor alternatives
Or only two good alternatives made
6 - 7 - 8
No
Are **no** alternatives stated or is there a **balance** of good and poor alternatives?
Yes
No alternatives suggested or a balance of good and poor alternatives stated.
5
No
Are **most** alternatives stated unrealistic, invalid or inappropriate?
Yes
A majority of poor alternatives outweighing some good alternatives
Or only two poor alternatives suggested
2 - 3 - 4
No
Are **all** alternatives (3 or more) stated unrealistic, invalid or inappropriate?
Yes
3 or more poor alternatives and no good alternatives?
1

Figure D.12 Q3.4a Assessment of implementation alternatives suggested by participants

1.14 Assessment of Participants' Evaluation of Implementation Alternatives (Q3.4b&c)



Evaluate the participants' own assessments of the impact of implementation alternatives they have suggested

Q3.4b In general, to what extent do you agree with the participant's assessment of the impact their implementation alternatives will have on the overall usability of ELDER?

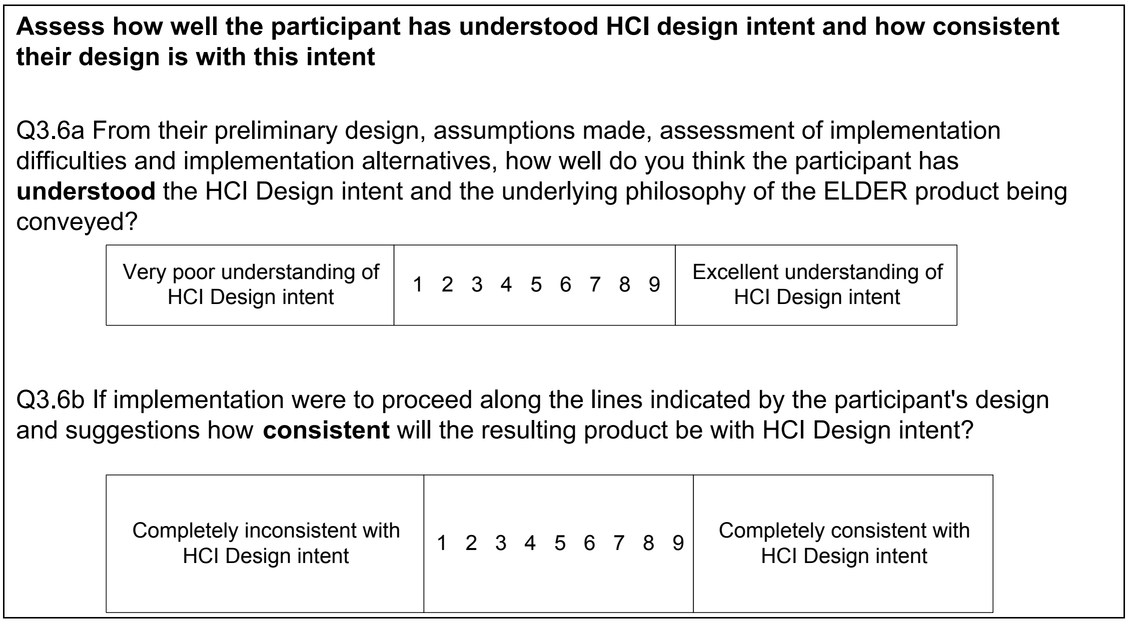
Strongly disagree with the participants usability assessment of implementation alternatives suggested.	1 2 3 4 5 6 7 8 9	Strongly agree with the participants usability assessment of implementation alternatives suggested.

Q3.4c In general, to what extent do you think that implementation alternatives suggested will assist the implementation of ELDER? (e.g. will alternatives allow the code to be structured better, simplified or will they reduce the coding effort required)

Implementation alternatives suggested will have a negative effect on implementation	1 2 3 4 5 6 7 8 9	Implementation alternatives suggested will have a strong benefit on implementation

Figure D.13 Extent of expert agreement with participants' own evaluation of the impact that implementation alternatives will have on usability (Q3.4b) and implementation (Q3.4c)

1.15 Assessment of Participants' Apparent Overall Understanding of HCI Design Intent and the Consistency of their Design with HCI Design Intent (Q3.6a&b).



Assess how well the participant has understood HCI design intent and how consistent their design is with this intent

Q3.6a From their preliminary design, assumptions made, assessment of implementation difficulties and implementation alternatives, how well do you think the participant has **understood** the HCI Design intent and the underlying philosophy of the ELDER product being conveyed?

Very poor understanding of HCI Design intent	1 2 3 4 5 6 7 8 9	Excellent understanding of HCI Design intent

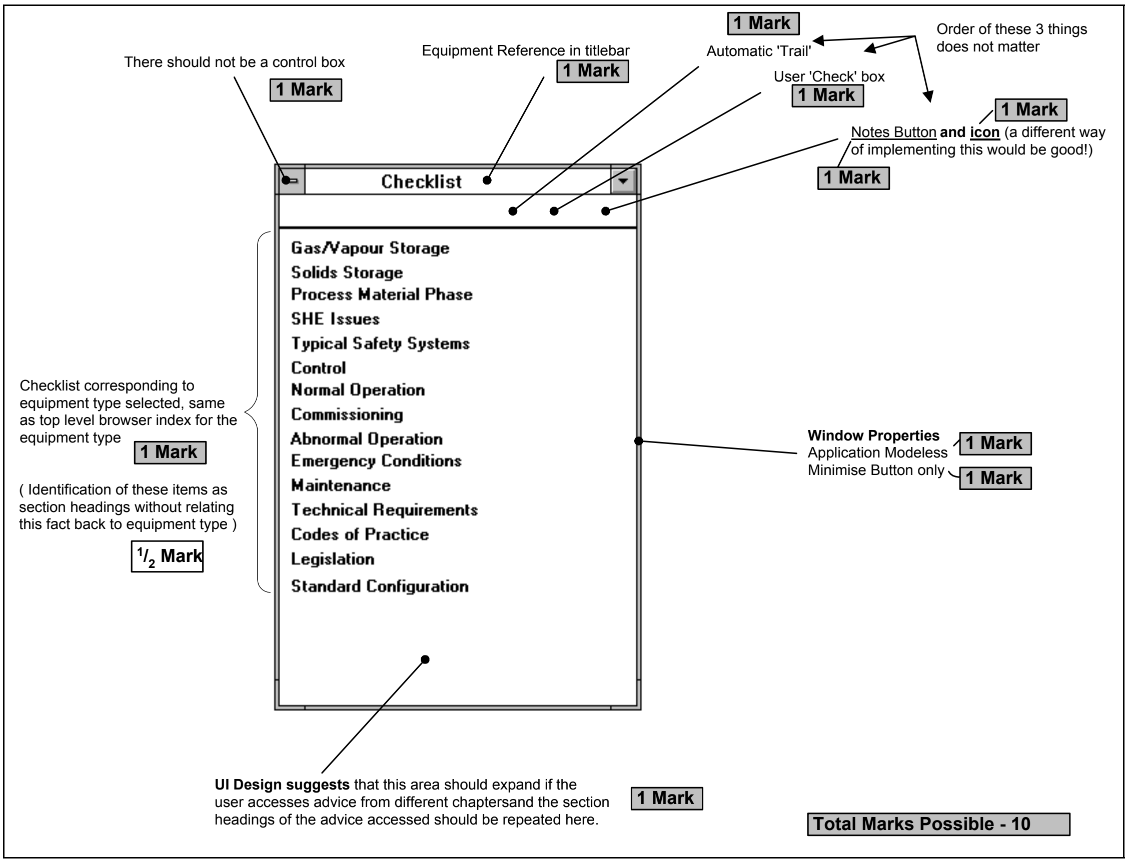
Q3.6b If implementation were to proceed along the lines indicated by the participant's design and suggestions how **consistent** will the resulting product be with HCI Design intent?

Completely inconsistent with HCI Design intent	1 2 3 4 5 6 7 8 9	Completely consistent with HCI Design intent

Figure D.14 Expert assessment of how well the participant appears to have understood the HCI design intent (Q3.6a) and how consistent their design is with HCI design intent (Q3.6b).

1.16 Framework for Quantitative Measurement of the Correctness and Completeness of Participants' Annotation of the Checklist Window in the Post-Test (Q4.1)

The pre-defined 'marking' framework for quantitative measurement of participants annotations of the Checklist Window in the post-test questionnaire is shown in figure D.15.



There should not be a control box 1 Mark

Equipment Reference in titlebar 1 Mark

1 Mark Automatic 'Trail'

User 'Check' box 1 Mark

Order of these 3 things does not matter

Notes Button **and** icon (a different way of implementing this would be good!) 1 Mark 1 Mark

Checklist corresponding to equipment type selected, same as top level browser index for the equipment type 1 Mark

(Identification of these items as section headings without relating this fact back to equipment type) 1/2 Mark

Window Properties
Application Modeless 1 Mark
Minimise Button only 1 Mark

UI Design suggests that this area should expand if the user accesses advice from different chaptersand the section headings of the advice accessed should be repeated here. 1 Mark

Total Marks Possible - 10

Figure D.15 Pre-defined 'marking' framework for quantitative measurement of participants annotations of the Checklist Window in the post-test questionnaire (Q4.1)