

COMPONENT TECHNOLOGIES AND THEIR IMPACT UPON SOFTWARE DEVELOPMENT

by

ANDREW DAVID PHIPPEN

B.Sc.(Hons)

A thesis submitted to the University of Plymouth
in partial fulfilment for the degree of

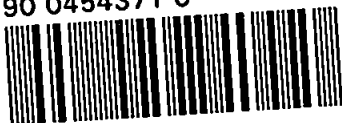
DOCTOR OF PHILOSOPHY

School of Computing

Faculty of Technology

November 2000

90 0454371 0



UNIVERSITY OF PLYMOUTH	
Item No.	9004543710
Date	- 1 MAR 2001 7
Class No.	T 005.1 PHI
Contl. No.	X40422071
LIBRARY SERVICES	

REFERENCE ONLY

LIBRARY STORE

Abstract

Component Technologies and Their Impact upon Software Development

Andrew David Phippen

Software development is beset with problems relating to development productivity, resulting in projects delivered late and over budget. While the term software engineering was first introduced in the late sixties, its current state reflects no other engineering discipline. Component-orientation has been proposed as a technique to address the problems of development productivity and much industrial literature extols the benefits of a component-oriented approach to software development.

This research programme assesses the use of component technologies within industrial software development. From this assessment, consideration is given to how organisations can best adopt such techniques. Initial work focuses upon the nature of component-orientation, drawing from the considerable body of industrial literature in the area. Conventional wisdom regarding component-orientation is identified from the review. Academic literature relevant to the research programme focuses upon knowledge regarding the assessment of software technologies and models for the adoption of emergent technologies. The method pays particular attention to literature concerning practitioner focussed research, in particular case studies. The application of the case study method is demonstrated.

The study of two industrial software development projects enables an examination of specific propositions related to the effect of using component technologies. Each case study is presented, and the impact of component-orientation in each case is demonstrated. Theories regarding the impact of component technologies upon software development are drawn from case study results. These theories are validated through a survey of practitioners. This enabled further examination of experience in component-based development and also understanding how developers learn about the techniques.

A strategy for the transfer of research findings into organisational knowledge focuses upon the packaging of previous experience in the use of component-orientation in such a way that it was usable by other developers. This strategy returns to adoption theories in light of the research findings and identifies a pattern-based approach as the most suitable for the research aims. A pattern language, placed in the context of the research programme, is developed from this strategy.

Research demonstrates that component-orientation undoubtedly does affect the development process, and it is necessary to challenge conventional wisdom regarding their use. While component-orientation provides the mechanisms for increased productivity in software development, these benefits cannot be exploited without a sound knowledge base around the domain.

Table of Contents

ABSTRACT.....	III
TABLE OF CONTENTS.....	IV
TABLE OF FIGURES	XII
ACKNOWLEDGEMENTS.....	XV
1. INTRODUCTION AND OVERVIEW	1
1.1 Introduction	1
1.2 Aims and Objectives of the Research	2
1.3 Thesis Structure.....	4
2. COMPONENT BASED SOFTWARE DEVELOPMENT – AN OVERVIEW ...	8
2.1 No Silver Bullets	10
2.2 Construction from Parts.....	11
2.3 Components and Component Standards.....	13
2.4 Defining Components and Component Standards	14
2.4.1 Scripting Components	16
2.4.2 Examples of Component Standards.....	17
2.5 Component-oriented Development - Why Now?.....	24

2.5.1	Technological Evolution.....	24
2.5.2	Management Appeal.....	28
2.5.3	Market Forces.....	32
2.6	The Philosophy of Component Orientation	36
2.7	Chapter Summary.....	37
3.	ASSESSING AND ADOPTING SOFTWARE TECHNOLOGIES	38
3.1	How Can We Assess a New Technology?	39
3.1.1	Defining Empirical Software Engineering.....	40
3.1.2	Techniques for Empirical Study	42
3.1.3	Criticism of Software Engineering Research.....	47
3.1.4	Software Process Research.....	53
3.2	Adopting Software Technologies	62
3.2.1	Diffusion of Innovations.....	64
3.2.2	Network Externalities	73
3.2.3	Organisational Learning	77
3.3	Summary.....	85
4.	A CASE STUDY BASED APPROACH TO THE ASSESSMENT OF COMPONENT TECHNOLOGIES	86
4.1	A Review of Case Study Research	90
4.1.1	The Research Design	91
4.1.2	Types of Case Study Designs	91
4.1.3	Data Collection.....	92
4.1.4	Data analysis.....	93

4.2	Relating the Case Study Approach with the Research Method	95
4.2.1	Defining the Research Approach in terms of Case Study Research	97
4.2.2	Data collection techniques	98
4.2.3	Data analysis techniques	99
4.2.4	Case Study Reporting	100
4.2.5	External Validity and Reliability in the Research Method	101
4.3	Summary	103
5.	SOFTWARE COMPONENTS IN THE TELECOMMUNICATIONS DOMAIN	
	104	
5.1	An Overview of the DOLMEN Project	104
5.1.1	DOLMEN Organisation Structure	107
5.1.2	The Use of Component Technologies in DOLMEN	110
5.2	The DOLMEN Case Study	112
5.2.1	Case Study Definition	112
5.2.2	Case Study Propositions	113
5.2.3	Case Study Role	113
5.2.4	Analysis approach	114
5.2.5	Case Study Review	116
5.3	The DOLMEN Software Development Process	117
5.4	The DOLMEN Component Platform	121
5.5	Case Study Analysis	124
5.5.1	Development Review	125
5.5.2	Trial Review	129
5.5.3	Reviewing the Results and Goals of DOLMEN	134

5.5.4	DOLMEN as a Component-oriented Software Project	134
5.5.5	Learning from the DOLMEN Experiences.....	139
5.6	Consideration of Findings Against Case Propositions	145
5.6.1	Proposition 1	145
5.6.2	Proposition 2.....	146
5.6.3	Proposition 3.....	146
5.6.4	Proposition 4.....	147
5.7	Chapter Summary.....	147
 6. THE USE OF COMPONENTS IN THE NETWORK MANAGEMENT		
DOMAIN.....		149
6.1	An Overview of Netscient Ltd.	149
6.1.1	Netscient Organisational Structure	150
6.1.2	Product vs. Domain Orientation	151
6.2	The Netscient Case Study	153
6.3	Case Study Definition.....	153
6.3.1	Case Study Propositions	154
6.3.2	Case Study Role	154
6.3.3	Analysis approach	155
6.3.4	Case Study Review	156
6.4	The Netscient Software Development Process	157
6.4.1	Netscient Domain Modelling.....	161
6.5	The Netscient Software Platform.....	167
6.6	Case Study Analysis	174

6.6.1	In-house Personality Management.....	174
6.6.2	Development Review.....	179
6.6.3	Issues Arising from the Use of Component Technologies	183
6.7	Consideration of Findings Against Case Propositions	187
6.7.1	Proposition 1.....	187
6.7.2	Proposition 2.....	189
6.7.3	Proposition 3.....	189
6.7.4	Proposition 4.....	190
6.7.5	Proposition 5.....	191
6.8	Summary	191
7.	PRACTITIONER SURVEY	194
7.1	Survey Approach.....	195
7.2	Survey Construction	196
7.2.1	Question Construction	197
7.3	Survey Response.....	198
7.4	Survey Analysis	199
7.4.1	Regarding your use of component technologies – establishing respondent type.....	199
7.4.2	Regarding your learning of component technologies – establishing learning approaches and common problems.....	208
7.4.3	Regarding component technologies and the software development process	215
7.5	Implications of Survey Results on Case Study Findings	234
7.5.1	Case personnel responses	234
7.5.2	Comparison of Responses Against Case Study Propositions	236

7.6	Chapter summary	238
8.	ADOPTING AND USING COMPONENT TECHNOLOGIES	240
8.1	Developing Case Study and Survey Results	241
8.2	A Reference Model for Component Platforms	242
8.2.1	Component Platforms	242
8.2.2	A Reference Model for Component Platforms	242
8.2.3	Current Standard Component Platforms	248
8.2.4	An Alternative Viewpoint – Visual Basic 3	254
8.2.5	Applications of a Reference Model for Component Platforms	255
8.3	Developing the Organisational Learning (OL) Perspective.....	256
8.3.1	The Organisational Learning Process	257
8.4	Approaches to Adoption	261
8.4.1	Standards/Guidelines	261
8.4.2	Transfer Packages	262
8.5	Pattern Approaches.....	265
8.5.1	Examples of Patterns	267
8.5.2	Consideration of Patterns from an OL Perspective.....	271
8.6	Conclusions: An Overall Strategy for Results Development.....	272
8.6.1	Refinement Based upon Industrial Feedback	273
8.6.2	Package Structure	275
8.7	Summary.....	276

9. A STRATEGY FOR THE SHARING OF EXPERIENCE IN THE ADOPTION AND USE OF COMPONENT TECHNOLOGIES	278
9.1 A Transition Package for the Adoption and Use of Component Technologies.....	278
9.1.1 Target Audience	279
9.2 Context	279
9.2.1 Reference Model for Component Platforms	280
9.2.2 Case Study Points of Reference.....	280
9.2.3 Survey Points of Reference	287
9.3 Language.....	287
9.3.1 Pattern template	288
9.3.2 Pattern Relationships	288
9.3.3 Patterns for the adoption and use of component technologies.....	290
9.4 Summary	330
10. CONCLUSION	331
10.1 Research Achievements	332
10.2 Research Limitations	334
10.3 Future Work.....	335
10.4 Technology Review	336
11. REFERENCES	338
A. DOLMEN BROCHURE.....	352

B. DOLMEN EVIDENCE EXAMPLES	357
C. NETSCIENT EVIDENCE EXAMPLES	366
D. COMPONENT SURVEY.....	376
E. SURVEY RESPONDENTS	380
G. PAPERS AND PRESENTATIONS.....	382

Table of Figures

Figure 3-1 – The Relationship between ESE, models and research questions	41
Figure 3-2 - Factors affecting product quality	54
Figure 3-3 – The Process Improvement Process.....	55
Figure 3-4 – The IDEAL Model	56
Figure 3-5 – Level of adoption based upon type of adopter	66
Figure 4-1 - Roadmap of Research	89
Figure 4-2 – Basic types of designs for case studies	92
Figure 4-3 – The process of developing theory from case studies.....	102
Figure 5-1 - An Illustration of an Integrated Services Environment	105
Figure 5-2 - DOLMEN Workpackage Structure	108
Figure 5-3 –DOLMEN Trial Set-up	110
Figure 5-4 - The DOLMEN Software Development Process	117
Figure 5-5 - DOLMEN Component Platform.....	122
Figure 5-6 – A sample MSC showing component interfaces and interactions between them (taken from [59]).....	127
Figure 6-1 - Netscient Organisational Structure	150
Figure 6-2 - The Netscient Software Development Process.....	158
Figure 6-3 – The Core Network Planning and Design Process	163
Figure 6-4 – Definition of Netscient Domain Functionality.....	167
Figure 6-5 - The Netscient Software Platform.....	169
Figure 6-6 – Netscient In-house Application Structure	174
Figure 6-7 – Netscient Personality Browser	176
Figure 6-8 – Netscient Personality Extractor.....	176
Figure 6-9 - Typical complex information structures in DOLMEN.....	185

Figure 7-1 - Component technologies used	201
Figure 7-2 - COM & CORBA related experience among respondents	202
Figure 7-3 - Experience classification of respondents	203
Figure 7-4 - Use of components in different project types	204
Figure 7-5 - Use of component in vertical sectors	205
Figure 7-6 – Determining an experience rating	207
Figure 7-7 - Distribution of experience ratings among respondents.....	208
Figure 7-8 - Learning about component technologies	209
Figure 7-9 - Problems when learning about component technologies.....	210
Figure 7-10 - Problems when learning component technologies.....	210
Figure 7-11 - Was the literature useful when learning.....	212
Figure 7-12 - Would it be useful to learn from the experience of others.....	213
Figure 7-13 - Was integration straightforward?.....	215
Figure 7-14 – Component-orientation makes software [easier, harder, neither easier or harder]?	216
Figure 7-15 – Willingness to use component technologies	219
Figure 7-16 - Comparison of component use with opinion regarding component difficulty	220
Figure 7-17 - Component technologies can be easily adopted	222
Figure 7-18 - Ease of adoption vs. integration problems.....	223
Figure 7-19 - Component technologies can be adopted independent of organisation issues.....	224
Figure 7-20 - Comparing agreement with question 18 against use of technology.....	225
Figure 7-21 - Project management is unaffected by component technologies	226
Figure 7-22 - Component orientation makes software reuse easy	227
Figure 7-23 - Ease of reuse compared to technology experience	228
Figure 7-24 - Component orientation should focus on software reuse	229

Figure 7-25 – Using Component Technologies is Straightforward	230
Figure 7-26 - Component orientation encourages design	231
Figure 7-27 - Component based development makes system deployment easier.....	231
Figure 7-28 - Ease of deployment against technologies used	232
Figure 7-29 - Component orientation makes system maintenance easier.....	233
Figure 8-1 – Original Component Architecture Reference Model taken from [68].	244
Figure 8-2- Reference Model of a Component Platform	244
Figure 8-3 – Mapping the OMA to the Component Platform Reference Model.....	249
Figure 8-4 – Mapping the Windows DNA to the Component Platform Reference Model	251
Figure 9-1 The component platform used in Case Study A.....	282
Figure 9-2 –The development process used in Case Study A.....	283
Figure 9-3 - The software platform used in Case Study B.....	285
Figure 9-4 –The development process used in Case Study B.....	286
Figure 9-5 – Pattern relationships	289

Acknowledgements

My thanks go to:

Chris, Steve & Peter - thanks for the supervision!

All at the NRG, probably the most entertaining office in the world.

My parents, for their constant support.

Ruth, for putting up with years of academic temperament.

And Matt, Charlie, Di, Mel, Bags, Vron, Bev, Jo, Andy, Chris and anyone else that keep me supplied with entertaining email throughout the writing of this thesis!

DECLARATION


At no time during the registration for the degree of Doctor of Philosophy has the author been registered for any other University award.

This study was financed with funding from the EU ACTS project DOLMEN, a studentship from the School of Computing, and industrial funding from Netscient Ltd.

Relevant conferences and DOLMEN project meetings were regularly attended (at which work was frequently presented) and a number of external establishments were visited for consultation purposes. Details of publications and presentations carried out during the research programme can be found in the appendices.

Signed

Date


15/2/01

This chapter introduces the concept of component-orientation, demonstrates the origins for the research programme and defines the aims and objectives. A discussion of thesis structure is included to introduce the reader to the various aspects covered throughout the text.

1. Introduction and Overview

1.1 Introduction

The ideal of component-based development is that application development becomes an assembly process, built on substantial reuse of standard components. In theory, more than 95% of an application can be based on reused software. [87].

Component-based techniques represent an area considered state of the art in software development. Numerous industrial sources [87, 28, 36,38] extol the virtues of a component-based development, and propose it as *the* technique that will enable software engineering to become a true industrial process.

The concept behind component-based development is straightforward – a software component represents an encapsulated piece of functionality that is reused at a *binary* level. This means that the reuse of each software component is implementation independent – one of the primary differences between component-orientation and other software reuse techniques. The evolution of software systems through component-orientation moves software development from engineering from first principles toward the systems assembly with reusable components. Theoretically, this should result in large increases in development productivity, as a greatly reduced amount of the system has to be written with original code.

This assembly technique reflects the industrial process in other engineering disciplines. There are often quoted comparisons between software engineering and, for example, electronic engineering.

Cox [41] highlighted the fact that, while the electronic engineer will achieve their requirements through the design and assembly of the electronic system using existing components, the software engineer will craft a system by creating new elements. This is equivalent to an electronic engineer starting from basic binary switches for any digital electronic system.

It is argued [42], that until software reuse becomes the standard technique for implementation, and the focus of development moves from programming to design, software development cannot be considered an engineering discipline. Previous software reuse techniques (for example, modular programming, object-orientation) have all been proposed as ways to increase development productivity, but have all fallen short of widespread adoption.

1.2 Aims and Objectives of the Research

This research programme aims to review the nature of software development and the concept of component-orientation, and to assess the impact of component technologies upon the software development field. In assessing the effect of component-orientation upon software development, it was intended that results would provide evidence for their potential, and also highlight areas of possible difficulty. As a development of these results, guidance could be provided for the future use of such techniques.

In the Joint IEEE Computer Society/ACM Software Engineering Body of Knowledge (SWEBOK) project, the publication of their report in Stoneman version 0.5 [35] stated in the area related to software infrastructure that:

Using components affects both methods and tools but the extent of this effect is currently difficult to quantify.

An interesting development of this point can be seen in the most recent version of the report [152], where the concept of component integration has been removed. The reason given for this removal was:

The editorial team concluded that while there was a strong industrial need for this type of knowledge [component infrastructure], there is not yet sufficient consensus on what portion of it is generally accepted. (pp. E-2)

Thus, the SWEBOK project, tasked with defining a core body of understanding for software engineering, has identified a need for the sharing of knowledge based upon component orientation, but has found it difficult to specify the nature of the knowledge required. This is very relevant to the research reported here, in which an important issue was to identify how best to assess component orientation and how to develop the result from that assessment into a usable form. This research project has identified specific areas for the reinforcement of knowledge in the area of component orientation, addressing the issue identified by the SWEBOK reports.

While the overall goal was to assess the impact of component technologies upon software development, several preliminary objectives were needed to place it in context. Specifically, the research programme sought to:

1. review the problems of software development, in particular, development productivity, drawing comment from leading texts in the area and examining the emergence of component-orientation as a development technique;
2. review literature in the area of software technology assessment and adoption, focusing upon empirical software engineering, software process improvement and theories of adoption as background research to guide the research in this programme;

3. gain practical experience in the use of component technologies within real world software projects, in order to assess the effect of component-orientation on software practice.
4. formulate theories in the use of component technologies, drawing from practitioner focused research, and considering the theories against popular beliefs regarding component orientation;
5. validate those theories through testing against practitioner experience;
6. review techniques for the transfer of software technologies into practice, in order to determine the most suitable approach for transferring experience from case studies;
7. formulate methods, based upon the above review and the findings of the research programme, to aid practitioners in adopting component techniques into their development approaches.

In achieving these objectives, the research programme would advance the state of the art in software development by providing novel contributions in terms of:

1. empirical evaluation of component-orientation in real world contexts, the outcome of the assessment being theories in the use of component technologies;
2. validation of these theories against the experiences of component practitioners;
3. formulation of methods to aid in the sharing of experience in the use of component technologies involving:
 - a reference model for component platforms;
 - the specification of an appropriately structured pattern language.

1.3 Thesis Structure

Chapter 2 examines component-orientation by tracing its origins in software reuse through to the standards and services that make up present day component technologies, and considers the

overall philosophy in the context of the universal software-engineering problem of development productivity.

Chapter 3 describes the aims of the research programme and reviews literature relevant to these aims. Initial consideration is made of research into the assessment of software technologies, both independently and as part of software process improvement techniques. Literature relating to the adoption of software technologies is reviewed, focusing in particular upon theories of technology adoption.

Based upon this literature review, Chapter 4 defines the research method for the programme. It further considers literature relating to the use of case study methods for the assessment of software technologies, and its application to the research programme, validity & reliability. The chapter concludes by defining and discussing the role of a practitioner survey to strengthen the external validity of results.

The next two chapters describe case studies used to assess the impact of component-orientation in industrial software, and have similar structures. The projects are introduced, and their aims and the role component technologies played in achieving them are discussed. The case study method in each project is described, defining case propositions, sources of evidence and data analysis techniques. Each case study's distinctive approach to component technology is reviewed, and issues regarding the use of component technologies identified. These issues serve as the basis for findings against case study propositions. This leads to a more detailed review of the achievements in each case, and considers the impact of component technologies upon the outcome. Conclusions are drawn regarding the use of component technologies in general.

The two case studies feature very different approaches to the adoption and use of component technologies. DOLMEN (chapter 5) was a product-oriented project within the telecommunications domain. In DOLMEN, a lot was assumed of the component-oriented approach, which was not borne out in reality. The Netscient project (chapter 6) took a more considered approach in applying component technologies, and was, in many ways, more successful.

Chapter 7 concludes the data collection aspect of the research programme by considering the theories developed from the case studies against a practitioner survey. This aspect of the research programme enabled the comparison of findings from case studies against the experiences of other leading edge software developers. This enabled a distinction to be made between exceptional phenomena and common experience from the studies. The survey also helped clarify the nature of the guidance practitioners need, and how it might best be presented.

Chapter 8 presents a strategy, based on the research findings to aid in future learning about, and adoption of, component-orientation. As a direct outcome from case study research and survey results, a reference model for component platforms that is used both as a means of comparison and a learning tool within the research programme is defined. The chapter continues by returning to adoption theories, identifying key concepts in the learning of new technologies. It goes on to consider existing approaches to technology transfer and determines the suitability of these approaches against both theory and the type of results from the research programme. A pattern approach is identified as the most suitable vehicle, and the chapter ends by considering the strategy for development using such a technique.

Chapter 9 describes a “transition package” that defines both a context and language for the learning of component-orientation. The context element uses the reference model of component platforms as a way of providing a technology independent view of component-based development and also as a means of comparing case study evidence. The context element also defines the nature of evidence that contributes to the pattern language by describing both case studies and also the practitioner survey. The context element thus strengthens the transferability of the pattern language. The remainder of the chapter defines the pattern language element, including specification of a pattern template and an illustration of pattern relationships. The patterns themselves are presented as problem/solutions pairs, reinforced with anecdotal evidence from the research programme.

Chapter 10 reviews the research method and discusses the main achievements of the programme. It also discusses limitations of the research to date and suggests possible future directions for the work. The concluding remarks return to the impact of component technologies upon software development in general.

The thesis also includes a number of appendices containing data to support the discussion in the chapters described above.

Finally, this thesis acknowledges the fast moving state of the field it assesses. As such, the component technologies discussed herein represent only a snapshot of the state of the field, from 1995-1999. More recent developments are not covered, as these could not be empirically assessed within the research programme.

This chapter is the first of three that considers bodies of knowledge relevant to areas within the research programme. The concept of component-orientation is discussed in greater detail. The material is intended to introduce the reader to issues in component-based development and discusses current thinking related to the component-orientation. The lack of academic literature is noted. The chapter also draws together a lot of discussion from industrial literature in determining the “philosophy” of component orientation, a concept against which assessment findings are compared. It should also be noted that this chapter is not intended to be a compressive review of all technologies within the component-oriented field. The emphasis is on those technologies used in the case studies.

2. Component Based Software Development – An Overview

This chapter examines the nature of component-based software development, the technological focus for this research programme. The review considers the background of component-orientation before discussing its underlying philosophy and its development. It is very much centred on industrial literature, as it is essentially from the industrial domain that the technology has emerged. While its origins can be traced to the 1968 NATO conference on Software Engineering [108], it has been industrial innovation that has placed it at the forefront of software development.

There have been three great revolutions in computing technology during the last 50 years: the stored-program computer, high-level languages and component-level programming. Although working programmers are well aware of the last revolution, it seems to have escaped the notice of most everybody else....The revolution has already happened, and in the academic community, nobody came.

The above quotation is taken from a recent paper by Maurer [98], highlighting the lack of academic research in the area. While some research within the wider domain of Commercial Off The Shelf (COTS) research has embraced component technology (in particular work at the

Software Engineering Institute on Component Based Software Development & COTS Integration [65]), it is generally agreed that component-orientation is an industrially based innovation.

However, before considering the specifics of component-orientation, we should consider its origins within the field of software reuse and development productivity. The following quotations illustrate the underlying problems in software engineering:

There is a widening gap between ambitions and achievements in software engineering. The gap appears in several dimensions: between promises to users and performance achieved by software, between what seems to be ultimately possible and what is achievable now and between estimates of software costs and expenditures. The gap is arising at a time when the consequences of software failure in all its aspects are becoming increasingly serious. [108]

The average software development project overshoots its schedule by half; larger projects generally do worse. And three-quarters of all large systems are "operating failures" that either do not function as intended or are not used at all. [62]

Although the message is the same, there is actually almost 30 years between the first, taken from the 1968 NATO Conference on Software Engineering, and the second from an article in the Scientific American in 1994. Both are essentially referring to the often-quoted *software crisis*, the software industry's continual failure to meet software demand with quality software, on time and in budget. This issue is actually a compound of a number of different problems, which together make up the overall predicament [62]:

1. software demand always exceeds software supply – currently the productivity of software developers cannot keep pace with the demands on their services;
2. software project management generally falls short on cost and time estimates;
3. software quality is sometimes less than adequate.

2.1 No Silver Bullets

As the software crisis was first identified during the 1968 NATO Conference of Software Engineering [108], it might be hoped that the software engineers would have addressed the relevant issues. However, as the above quotations demonstrate, the same criticisms levelled at software development over thirty years ago can still be applied. In this time there have been numerous development techniques and improvements in information technology. However, there has also been a marked increase in the demand for software and the domains in which it is used. While development productivity has undoubtedly been greatly improved as a result of new development techniques and technologies, the increase has not matched the expansion in demand for software. In his seminal paper “No Silver Bullets: Essence and Accident in Software Engineering”, Fred Brooks [25] stated that there had been no software development technology that had introduced an order of magnitude change in developer productivity – a necessary increment if productivity will ever meet demand. If one considers the improvements in development technology that have occurred (for example, procedural programming and object oriented programming) it seems that they are simply techniques for improved implementation or coding - what Brooks refers to as solutions to *accidental* issues in software.

This differentiation between *accidental* and *essential* change in software development is the underlying message from Brooks’ paper. Brooks talks about the essence of creating software being the actual crafting of a conceptual construct into software form. If we look at how we *essentially* develop software - determine requirements, design, then implement, it is true to say that new techniques have caused no significant change in this approach.

Perhaps a central problem in the development of new ways in which to write software is that we are too focused on the actual *software development* aspect of the problem. The assumption is that software development is too slow / unreliable / etc. – that we must increase the speed at which we *write* software. As a result, we end up with better ways to do the same thing, which is, inherently, the wrong thing to do. There will never be an order of magnitude change in the productivity of software development if all we are doing is reinventing the same technique. Developers are losing sight of why software is developed – what is the software trying to achieve? What we must realise is that software is a service industry. It can only exist within another environment. This point is illustrated well by Grady Booch, in [23]:

Banks are in the business of managing assets; software is just a business tool for responding to those needs. Libraries are in the business of facilitating access to information; software is just a means to that end. Manufacturing companies are in the business of creating hard goods from raw materials; software is a kind of soft goods that makes that process more efficient and hence more profitable.

Software enables the delivery of information to a given user in a given way - software systems can be seen as processes that take, transform and present to the user, fulfilling their specific information requirements.

As already mentioned, the problem with the majority of “new technologies” is that what they provide is better ways to do the same thing. The focus should not be on implementation issues, but on how to model the business problems into software form.

2.2 Construction from Parts

In the follow up paper to “No Silver Bullets”, Brooks [26] suggests that software reuse *potentially* offers a way to greatly improve developer productivity, stating that the best way to attack the method of building software is to not build at all. The concept of software reuse has been around for as long as software engineering itself. The central idea is that much of what is coded into

software is similar each time. Therefore, instead of re-coding identical functionality, it would be far more sensible to reuse parts of software that have already been written. Cox [41] likens the recent state of the software industry to more of a crafting ethic, where each piece of software is individually created from scratch. In the same way that a craftsman would, for example, build a new table from core raw materials (wood), craft each piece of the table and then put it together, the software developer crafts a new application. Starting with the raw materials (source code), each aspect of the application is crafted before being integrated to making the application.

Cox argues that in order to achieve any major developments in software development productivity this craft ethic has to be changed. He views large-scale software reuse as the industrial revolution of the software world – finally moving from craft to industry. Object-orientation (OO), when it first emerged, was held up as the development technology that would enable this shift. Object-oriented programming languages, such as Smalltalk and C++ provided the programming constructs to build software with objects and classes. Object-oriented analysis and design techniques (for example, Object Modelling Technique [135]) provided similar techniques for the modelling of a system in an object-oriented form. Using these techniques, it was predicted that object-orientation would be an enabling technology in software reuse.

OO has now been a mainstream technology for ten years, and developers still face the similar productivity and management problems. We could conclude from this fact that OO has not fulfilled its potential. However, while it could be said that object-orientation has not achieved its full potential as a reuse technology, it has, through its development, been influential as an underlying technique for technologies such as visual programming and object frameworks. These *object technologies* (rather than specific object-orientation) undoubtedly enable far great developer productivity.

Component-orientation can be seen as a progression of object technology - embracing the idea of building software from *components* while attempting to avoid the pitfalls of pure object-orientation.

2.3 Components and Component Standards

Component-orientation and reuse can be considered to be the foundation of any mature engineering practice. However, the vast majority of software projects still have very little reuse. The concept of achieving requirements through the construction of pre-existing, or third party components seems to go against the ethos of the software development, which instinctively seems to be of the opinion “if you didn’t write it, don’t use it”.

The often-quoted origin of component software comes from a paper presented at the 1968 NATO conference on software engineering by McIllroy [100]. This means that, as concepts, software engineering and software components are of the same age. In his paper, McIllroy put forward the concept of a software component as a library of routines that can be reused in software applications through a standard interface. While this definition differs somewhat from what we would now consider a software component to be (see section 2.4), two issues were introduced that are still highly germane. In particular:

1. **The component market place:** The component marketplace extends the traditional model of purchasing software, which centres around the application as a single unit of sale, to incorporate components developed by a third party. The component market place is still seen by some (for example, Chappell [37]) as an essential part of the move to component-

orientation – it enables developers to focus on their own domains and purchase third party components for other aspects of the application they are developing.

2. **Standard ways to interface with components:** While McIlroy identified this need in 1968, the sort of standards to which he refers have only started to be available to the developer in the past five years. A standard way of interfacing components is essential for the component marketplace. Completely independent developers can write components to the same standard and be sure that their components will be able to interact.

The following section discusses component standards in more detail.

2.4 Defining Components and Component Standards

There is no agreement about the formal definition of the term *software component* (see, for example, [26], [21], [37], [34]). However, we can identify common aspects from these definitions:

- **It is a packaged piece of software, reusable in binary form independent of language or platform:** The central aim of a component-oriented approach to software development is to provide reuse at a binary level, not source code like previous development technologies (such as object-orientation). Total interoperability independent of language and platform through binary reuse can be considered the utopian aim of a software component. Case study experience (see chapters 5 and 6) has demonstrated that this aim is still not fully realised.
- **It exposes functionality and properties via interfaces:** The concept of interfaces is essential to the software component, as it is via interfaces that the component can be reused independent of language and platform. The interface provides a separation of defined functionality and actual implementation. The component client (i.e. the piece of code that calls component functionality) need only have access to a component's interfaces to be able

to exploit the functionality behind the interface. The component standard (see below) maps the call from the interface to the functional implementation.

- **It defines methods, properties and events:** Methods and properties map to the concepts of behaviour and state in object-oriented systems [22]. However, properties can extend the concept of state from the OO definition. Within an OO class, instance variables are defined to indicate the state of an object, using simple values. As a component's properties are exposed through an interface, they do not have to map to simple values as they can be used to dynamically realise state based on functional parameters. For example, a banking component that exposes a property called *balance* could map that property to a simple variable that holds a given balance. However, it could also map to some functionality to calculate the balance dynamically from other values, or from interfacing with a database. Events enable a component to communicate occurrences that could affect its external environment asynchronously, in a similar way to the event driven mechanisms that manage most windows systems. To use the banking example again, if a withdrawal made an account overdrawn, the component could fire an *overdrawn* event, which enables other system components to deal with this occurrence in an appropriate way.
- **It is written to an interaction standard:** The component standard provides a set of rules for the structuring and interaction of software components.
 - **Component structure – or the *component model*.** This defines a standard way for the component to be structured, such that developers, using development environments and containers, can access and use the component. Generally, a true component model structures the component in properties, methods and events, as described above.
 - **Exposing functionality and structure –** dealt with using interfaces, also discussed above.

- **Component containers** – In order to be of any use the component requires a *runtime environment* in which to exist. The runtime environment, or container, provides a context where components can be assembled and used. The containers could be applications (for example, Internet Explorer) or parts of an application (for example, a compound document comprising a Word text and an Excel spreadsheet). It is the role of the standard to define how the components are contained (for example, what interfaces a container expects a component to expose).
- **Component location and interaction** – the standard should also define how components are located and the protocols for interaction between them. This removes any need for low-level code in a component client to deal with component location or network communication. All that the client requires is a component reference that the standard can use to locate the component.

2.4.1 Scripting Components

A final aspect of componentware, whose importance has been demonstrated throughout this research programme, but is generally not included in the definitions of a software component, is that the component should be scriptable. One of the major arguments for the use of components in organisational software development process is that they provide a high level of reuse. The development of *componentware* - software constructed with components – with programming languages (i.e. C++, Java, Pascal, etc.) is effective, but generally still requires knowledge of the component standard. It is the use of very high-level languages (scripting languages) – Visual Basic for Applications (VBA), JScript, etc. - which provides the most effective means of rapidly constructing a complex application from reusable parts. Therefore, without a scriptable element to the component / component standard, it could be argued that the reuse potential for a component is not as high as it could be. To demonstrate the difference in complexity between programming

and scripting languages, the following are code fragments for the calling of a function on a COM class, firstly using C++, and then using VBA:

C++

```
#define CLSID_TESTSERVER 17CDF24E-8862-11D2-8A8C-
0060972FB3BF
...
HRESULT hr;
ITest *m_pTestInterface;

hr = CoCreateInstance(CLSID_TESTSERVER,
    NULL,
    CLSCTX_INPROC_SERVER,
    IID_ITest,
    (PPVOID)&m_pTestInterface);

if (SUCCEEDED(hr)) {
    hr = m_pTestInterface->TestFunction();
    if (SUCCEEDED(hr)) {
        ...
    }
}
```

VBA

```
Set myObject = CreateObject("Testdll.TestServer")
myObject.TestFunction
```

2.4.2 Examples of Component Standards

This section provides examples of component standards. It is intended as an introduction to each standard and discusses their differences, problems with use and future directions. It does not provide a comprehensive definition of the features of each standard - readers are referred to the numerous technical texts referenced below for more detail about each.

2.4.2.1 Microsoft COM/DCOM

Microsoft's Component Object Model (COM) [134] is currently the dominant component architecture, mainly because it resides in Microsoft's flagship operating systems and it is the foundation for all of the Microsoft application technologies (such as OLE and ActiveX). Distributed COM (DCOM) [64] extends the basic COM functionality to incorporate a transparent network distribution mechanism into the architecture.

A common criticism levelled at the COM approach to component software is its complexity. The COM standard defines various application services (for example automation (see below), compound documents, drag and drop, ActiveX controls, etc.) which use the COM standard as a platform. Each service specifies a number of interfaces that a component must implement in order to comply with the standard. This process is made more complex by the fact that some interfaces have to inherit from other standard interfaces in order to function in the correct way.

Microsoft acknowledges the complexity of the COM standard [34] but much is hidden from the developer through development environments. While development in an environment such as Visual C++ still requires that the developer is fairly knowledgeable about the COM standard, using Visual Basic (versions 5 or 6) isolates virtually all of the COM functionality from the developer.

The complexity of writing components in COM was further reduced with the introduction of Microsoft Transaction Server (MTS) [85]. While its name suggests a relationship to database transaction control, what it actually provides is a framework for the development of server components so that a developer can focus on implementing the business logic required in the server. Using MTS, all low-level component functionality required to cope with server side

processing is managed. To use the spell checker example, the client sends the server the word it wants to verify, the server then looks up the word in a dictionary, advises whether the spelling is correct, and if not, suggests alternatives. However, if a number of clients were all wishing to use the server at the same time, several problems arise. Firstly, each client requires an instance of the server to use. Then, every instance requires a connection to the dictionary resource, which could be a local file, or could be on a database. One can see that even with this simple example, a small number of clients would place a significant load on the server, and require the developer to incorporate scaling code (i.e. resource pooling, threading, etc.) into it. However, developing the server component within the MTS relieves the user of these problems. The code for the MTS server is essentially the same as a standard COM component (with a few calls using the MTS API), with all threading, resource pooling, security, etc. dealt with by the MTS framework. The MTS framework is, arguably, the most important piece of component technology to be introduced by Microsoft, as it provides such an effective wrapper around the COM standard.

As a final comment, a further extension to the COM architecture, COM+ [88], is included in the Windows 2000 platform. COM+ further extend the MTS model for components (i.e. write a single user component which can automatically be scaled to enterprise level), essentially providing another wrapper around COM. The COM+ “wrapper” provides functionality for such components services as asynchronous messaging, in-memory databases, self-describing components and attribute-based development (i.e. embedding simple notes in code which are used by the environment to configure the component at runtime - for example, whether it requires transactions, what levels of security it requires, etc.).

2.4.2.2 CORBA

CORBA (Common Object Request Broker Architecture) is an architectural specification by the Object Management Group (OMG) – a consortium comprising over 800 members. Its motivation was primarily to provide a standard for the distribution of *objects* over heterogeneous networks. Essentially, via a process of committee-based review, the OMG developed and released the CORBA standard. The first version was released in 1992 and following a major review, it now exists as version 2 [109]. A second major review should result in the CORBA 3 standard being released sometime in 2000¹. The OMG states that CORBA's strength lies in the functionality defined to allow the distribution of object solutions, and in its platform and language independence.

Unlike COM, CORBA exists solely as a specification, it is up to vendors (for example, Sun, Iona, Visigenic) to provide *ORB implementations* based on the specification. In theory this makes CORBA entirely independent of language or platform. The CORBA specification provides language mappings - directions for how a given language will implement a CORBA interface and the functionality required to realise that interface as an object. Vendors then work with the standards and mappings to develop implementations for whatever platform they wish. However, in reality the standard/implementation separation has led to many problems. While the theory of providing a standard is sound (i.e. everyone works to the same standard, therefore everything works together), the reality is that ambiguity in the standard has resulted in different CORBA implementations being unable to interoperate. This problem is compounded by vendors introducing new features, external to the CORBA standard, into their products. Therefore, what

¹ At the time of writing (September 2000) the CORBA 3 specification had not been publicly released by the OMG.

developers end up with is a choice of implementations based around, but not on, the CORBA standard.

As an attempt to address this problem, the OMG introduced the concept of inter-ORB protocols in version 2 of the CORBA specification. The most common inter-ORB protocol is IIOP (Internet Inter-ORB interoperability Protocol) that enables interoperability over Internet network protocols. However, this interoperability standard was, once again, a paper standard with no core implementation. Therefore, a similar problem to that encountered with standard implementations can occur with IIOP implementations. A controversial report by Ovum [132] discussed this issue in greater depth, concluding that pure CORBA products were of little use as they provided only functionality for a standard with no chance of interoperability with other implementation. The report stated that it would be the CORBA-based products, such as Iona Orbix and Inprise Visibroker that would be more successful, as long as an enterprise stayed with a single implementation. Certainly, developing objects using a single CORBA implementation, for example, Orbix, does provide good potential for object reuse at an enterprise level.

The development of CORBA systems is, generally, not as complex as developing pure COM components. Arguably, there is no component model (see the discussion at the beginning of section 2.4) on which to work. While the approach to exposing functionality is no different to COM (i.e. through interfaces), there are no standard interfaces for the developer to implement. The CORBA developer simply specifies an interface in the OMG Interface Definition Language (see [109]), implements the methods within a server class, and then binds the class to the interface in a server *process*. The choice of development language is still important for development productivity, as the language mappings add a layer of complexity to the core language. Therefore, writing CORBA objects in C++ is more complex than writing standard C++ objects. The

implementation also requires a good knowledge of the mapping itself, and the workings of the CORBA standard.

Another problem in the development of CORBA systems is the lack of development environment support. As a base standard, Microsoft COM is far more complex than CORBA. However, Microsoft wraps the complexity of the core COM implementation into its development products. Certain CORBA implementations (for example, the Inprise products that use the Visibroker technologies – see www.inprise.com/visibroker) do have inbuilt support. However, for the majority of ORB implementations, especially in the UNIX environment, the developer has to rely on text editors and command line tools to write the interface definitions, compile the interface definitions, write the server implementation and write the server process. Therefore, the reduction in complexity of the standard is offset by the complexity of the actual development process in implementing a CORBA object.

Other weaknesses of CORBA arise from the lack of a full component model:

- there is no standard way of implementing events in a CORBA object. However, events are supported in CORBA using the Event Service [110];
- the packaging of objects is restrictive – at present, in order to distribute CORBA objects, one has to provide a process (i.e. an executable application) which holds instances of the bound objects. This process has to be executed in order that clients can gain access to the server objects;
- there is no specification for CORBA object containers in the standard;
- CORBA objects are not scriptable – a Request for Proposals (RFP) by the OMG for a scripting model, resulted in a few attempts at making CORBA objects scriptable, for example [40]. However, due to the lack of standard in these scripting approaches, none have been

adopted by any mainstream component containers (e.g. Visual Basic, Internet Explorer, etc.).

The closest CORBA implementations have to scriptability is IIOP support in applications such as Netscape Navigator and Lotus Notes.

It is proposed that the CORBA 3 standard will provide a component model for CORBA objects [111], as well as a scripting model and pass-by-value features that should enable the simple passing of complex information structures. These new facilities will make CORBA a more complete component standard. However, as this research programme aims to empirically evaluate CORBA 2 and DCOM technologies only (as a result of the nature of the case studies), the CORBA 3 standard is not explicitly addressed in this thesis.

2.4.2.3 Sun JavaBeans

JavaBeans is a Java API produced by Sun which allows developers to write components based on an extended version of Java (incorporated in standard APIs in Java 1.1 and extended in Java 2 [94]). The JavaBeans API does not provide any in-built support for component distribution, but there is a growing trend to write distributed JavaBeans applications using CORBA. Additionally, another API included from JDK 1.1 is the Java Remote Method Invocation (RMI), which enables Java specific distributed communications. Finally, it is also possible (though slightly idiosyncratic) to distribute JavaBeans using DCOM.

While the JavaBeans standard provided core functionality to make Java classes into components, it lacked a great deal of the richness and power of CORBA or DCOM. More recently it has been adopted as the foundation component model for Enterprise JavaBeans (EJB) [154]. The original intention of EJB was to join Java and CORBA into a comprehensive standard for distributed, enterprise component-oriented system. However, as the standards developed, and as the

requirement for further componentisation in the CORBA 3 standard became apparent, it seemed logical to develop EJB concepts away from Java specifics to be used within the CORBA 3 standard. It is envisaged that the future of EJB is as the Java mapping for the CORBA 3 standard [74] and also as an element of the Java 2 Enterprise Environment [151]

2.5 Component-oriented Development - Why Now?

It has already been mentioned that the concept of component-oriented software development has existed for almost thirty years. However, significant interest in component-oriented systems has come about only in the last five years. This raises the question as to what aspects of the current software environment have enabled component software to finally move from theory into practice.

In the following we consider three pre-requisites of an effective environment for component-oriented software:

- Technological evolution;
- Management appeal;
- Markets.

2.5.1 Technological Evolution

Chappell [37] argues that the blossoming of component-orientation must be attributed to the evolution of both the software environment and development technologies. He defines eight key areas of technological growth.

1. *"Acceptance of a standard component model"* - this is certainly a crucial point in the technological drive to use component-oriented techniques. Without a standard by which to

write and construct components, the developer would have no way to interact with other components. With the emergence of standards, a developer knows that as long as their components adhere to that standard, any other component based on the standard should be interoperable, no matter how it was written.

2. *"A large third-party component market is in place"* – A criticism of the object-oriented approach is that in order to benefit from the reuse potential, it is first necessary to write a framework of objects for the organisational need. While some third party frameworks exist and are used successfully (e.g. Microsoft MFC), they still suffer from unclear interfacing and interoperability issues, and therefore have limited reuse potential (for example, in order to use MFC efficiently, one is tied to using Visual C++). With a truly effective standard, consumers can shop around for components that match their needs knowing that the imported building blocks can be integrated with their custom software. This opens the way for small software houses to specialise in marketing specialised components rather than trying to compete at the application level with large organisations. As discussed in section 2.3 the component marketplace was one of the major themes in McIllroy's seminal paper. However, it is only with the advent of effective component standards, almost thirty years after McIllroy's observations, that a component marketplace is being realised.
3. *"The types of component available are rapidly expanding"* – while the impetus for Microsoft's drive in the component field was, arguably, down to the unexpected success of visual (GUI) controls, both of the major forces in component technology development (Microsoft and the OMG) are focussing more on *vertical* encapsulation. This should result in the development of component suites for specific industries (e.g. healthcare, finance, telecommunications, etc.). Additionally, there are many component software houses starting up with domain-specific knowledge, enabling them to compete on a far smaller scale than if they were in the applications market. This progression in the development of third party,

domain-specific components provide application developers with many more possibilities for component reuse.

4. *"Components are moving off the desktop and beginning to play an important role in creating server applications"* – while the desktop (client-side) offers great reuse potential for GUI components, the move away from stand-alone applications means that a lot of the functionality of an application will exist on the server. The term *business objects* [113] is often used to describe this sort of component – a component that encapsulates a business entity in a non-application specific way. The reuse potential for business objects is extremely high and that potential can be increased even further with a component-based approach, as interfacing to the component is straightforward.
5. *"Components are a key part of web-based applications"* – web applications, due to the diverse, distributed environment in which they exist, tend toward implementation independent technologies based around agreed standards. The integration with component standards that already communicate using the same network protocols as the web (TCP/IP) enables a huge amount of functionality to be accessed via a standard web browser. Additionally, some web browsers already exist as component containers (for example, Internet Explorer) so developers can guarantee client-side functionality through wrapping in a component standard.

Chappell's final three points all related to developers and developer productivity, namely:

6. *"It has now become significantly easier to create components"*;
7. *"A critical mass of component-based developers exists"*;
8. *"Powerful tools have become available for designing and testing component-based applications"*.

While the points made are all valid, they are, perhaps, a little optimistic. It is certainly easier to create components with today's development environment than early development approaches. C and C++ were the primary ways to develop "first-generation" components (i.e. early CORBA, VBX, OCX, etc.) with, especially for CORBA, very little in the way of development environment. Newer environments (Visual Studio 6, Inprise Visibroker technologies, Iona's Orbix RAD product [75]) integrate the component standard effectively into the development environment and provide productivity tools for the *writing* of components.

The fact that it is now possible to develop components in the developer's "language of choice", rather than being tied to C or C++, aids in the productivity of component and component container production. At the very least, the fact that components can be developed in numerous different languages and environments means that the potential number of component developers is considerably higher.

Component configuration and distribution are two areas where the tools are still lacking. A large proportion of time spent implementing a component system will be in the configuration of the components in their environment (registering, distributing, configuring security, etc.). It is this *deployment phase* where knowledge about the workings of the component standard is more important than in the development phase. For example, an Orbix-developed CORBA object requires the construction of a server process that will be the "container" for the class. The developer then has to deploy the server in such a way that the ORB will know which server to start if a client request for the given object is required. This is either carried out using tools provided as part of the Orbix distribution or using an implementation of the CORBA Naming Service [110]. Only then is the developed object available to other clients in the distributed environment.

A truly effective component development environment would make this entire process, from development to deployment, transparent.

2.5.2 Management Appeal

As stated above, technological evolution is certainly not the only reason that component-based software development is currently gaining momentum, both as a hyped technology and as a realistic way of developing software. As a software technology, component-orientation can be marketed to appeal to software managers, who are, essentially, the people that need to be convinced if a new development technology is to be adopted.

The greatest pressure on software managers is to deliver quality software on time and on budget.

In order to achieve this, software managers are constantly looking for ways to:

- improve development productivity;
- increase the reliability of software;
- reduce maintenance overheads.

The following discusses the ways in which component-based development addresses each of these areas:

Development productivity – A primary argument for the use of component-based development is that software components, by their nature, can be effectively and easily reused in any number of container applications. Software reuse has long been regarded as one of the most effective ways to improve software productivity [82]. Previous attempts at technologies for reuse (modular

programming, object-orientation, etc.) have suffered due to problems with the way the provision for reuse is achieved. However, it should be acknowledged that, once again, it is only implementation that this aspect of component-orientation addresses – conventional wisdom (see section 2.6) does not consider reuse in other activities such as design.

Increased reliability – Firstly, the component technologies can provide a lot of the infrastructure (communication, security, etc.) need in distributed, enterprise away from the developer. Therefore, developers can focus on the implementation of a business problem. Use can also be made of components that have already been implemented and tested. Both greatly reduce the amount of new, untested code that needs to be developed for a new system.

Reduced maintenance – As a component-oriented application is constructed of parts, all of which are maintainable separately, the potential for a more flexible approach to maintenance increases. In the case of a component containing a bug, the bug can be fixed and the new version of the component can be plugged into the application without the rest of the application being affected.

Additionally to these traditional software management problems, the volatility and variability of the current and future software environment has placed a far greater demand on software houses to be rapidly reactive and flexible in their development approaches. The following identifies emerging areas to which software houses will have to adapt, and discusses the attraction and demand for each:

Heterogeneity – coupled with the demand for distributed processing, heterogeneity in a distributed system is an important development in the software environment. To use the WWW

as an example, when a user is browsing a web site, they are not aware of the nature of the server platform. To take another example, consider the telecommunications networking domain. In this domain the vast majority of low level work (i.e. interfacing management systems to the communications hardware) is carried out on UNIX systems as they tend to perform better at low level tasks than the equivalent PC environments. However, UNIX is notoriously complex for human-computer interaction. Even with user interface additions, invariably the user requires a good knowledge of the operating system in order to use the system effectively. A more desirable system would be to keep the low-level communications aspects of the systems on UNIX platforms and provide user interfaces to the systems on the more familiar, and user friendly, Windows-based PC environments.

Scalability – the concept of scaling a component system has already been discussed above (see section 2.4.2.1). Consider an application developed as either a stand-alone or simple client/server system as a demonstration prototype to refine user requirements. Once requirements are agreed, the system needs to be scaled to meet the demands of a huge multi-user system across an enterprise. Ideally, the prototype system could be scaled to accommodate these changes rapidly (the core functionality being the same). In reality, this invariably means a complete rewrite and the prototype would not be able to cope with the demands of a large-scale distributed system.

A component-based approach introduces a great deal of flexibility in addressing all of the above problems. With respect to heterogeneity, this is something implicit in component-orientation. A component client needs only the interface definition and reference to be able to call services from a given server component. The client neither has, nor needs, an awareness of the server implementation (in terms of both platform and development language). Heterogeneity is a driving force behind the CORBA standard – a reason for producing a standard rather than an

implementation was that vendors could implement for whatever platforms they wished. Certainly, CORBA implementations exist for all major UNIX implementations, Windows platforms and a few embedded and mainframe systems. Therefore, it is entirely possible to build distributed systems in a heterogeneous environment with CORBA, exploiting the platform benefits at each node of the system. As mentioned above, DCOM, as a component standard implicitly supports heterogeneity. However, at the current time, UNIX implementations of DCOM are limited (although Software AG's EntireX technology [143] has developed DCOM implementations for a number of UNIX platforms).

Scalability is not addressed directly by each of the core standards. However, additional services are beginning to deal with these issues. The COM/DCOM standard is greatly enhanced for this purpose with the MTS product, which essentially removes the vast majority of scaling issues from the developer. COM+ is intended to develop these services further, to eventually make it as easy to deliver enterprise applications as it is to deliver workgroup applications. [88]

For the CORBA standard, a number of services exist to deal with scalability (e.g., transaction control, security, events and messaging). As with everything CORBA-related, it is up to the CORBA implementation vendors to realise these services into products.

The above demonstrates the appeal of component-oriented systems to software managers. Not only do they help in achieving the traditional tasks of the software manager, they also enable an organisation to be highly dynamic in meeting the demands for more complex and flexible software.

2.5.3 Market Forces

The final aspect of influence over the drive toward component-oriented development is the often-overlooked aspect of market pressures exerted by the people that produce the standards.

Consider the following: software developers require development tools in order to make the mundane aspects of their task as straightforward as possible. They have to purchase these tools from a vendor. Hence, development environment vendors have a lot of power in influencing the way in which we develop software. While we may feel that we have a free choice in our selection of development products, we work in a highly volatile industry where certain skills can count for considerable earning potential. Therefore, we are drawn to the technologies most desirable by organisations and recruitment agencies. For example, a few years ago Borland and Microsoft battled for the dominant C++ development environment for Windows. Initially the main conflict was between which was the better object framework, OWL or MFC, and which had better “visual” capabilities. However, the eventual dominance of MFC and Visual C++ had little to do with the products and more to do with the greater strength Microsoft had at marketing the product and, therefore, creating demand for MFC/Visual C++ skills. As these skills became desirable, developers felt obliged to use the product in order to be more employable. We can view this as an example of a way in which the way we develop software was essentially dictated to us by a corporation.

In the component field, there are two dominant forces influencing the directions in which component-orientation could be taken:

1. Microsoft, whose technologies include OLE, MTS, COM+, etc. (although all are based on COM/DCOM)

2. The Anti-Microsoft Lobby (CORBA, JavaBeans, Enterprise JavaBeans, etc.), composed of companies such as Netscape, Sun, Iona, etc.

It can be argued that the first major impact of component-based technology occurred with the advent of Microsoft's VBX controls [46] for Visual Basic. VBX (Visual Basic eXtensions) controls provided developers with an SDK to write their own visual controls (specifying properties and events and implementing behaviour based on an event model), which could be plugged into Visual Basic's development environment and used in the same way as any of the standard controls. While the VBX model itself was limited (it was 16 bit, the component model was incomplete, there was a single C-based SDK) the interest they generated caused Microsoft to push for a more stable model based on their OLE technologies. The eventual outcome of this momentum was the standardisation on COM as an underlying technology for all of application communication mechanisms (DDE, OLE, OCXs, etc.). Microsoft claims that this convergence was the result of fifteen years development of technologies to realise that standard [96].

Microsoft currently holds the larger market share in the component marketplace. Any user working on 32 bit Windows platforms is unavoidably using components as virtually every Microsoft product released is component-oriented (the most widely used example being the Office suite, which has been component-based since the Office '95 release). The two most recent releases of the Visual Studio suite have been progressively more component-oriented. One can imagine further releases providing even more component support. Consideration should be given as to what Microsoft's motives are for virtually forcing users and developers down a component-oriented route. Admittedly, component-orientation promotes good software reuse and maintenance practices. It also vastly increases the potential for third-party reuse and rapid application development.

Microsoft currently stands on the verge of a monopoly in the PC market. They hold nearly 90% of the desktop market [146]. However, they are very much tied to the Intel/PC world, as there are few ports of their operating systems to other hardware platforms. Therefore, there is still a large section of the IT world that is not dominated by Microsoft (particularly the server side). Explosions in distributed computing - in particular the Internet, but also using distributed architectures – have meant that there is even less need for servers to be the same type as desktop clients. While trying to compete on an operating system level would require the porting of huge amounts of code to a different hardware type, an alternative could be to work above the operating system. One of the benefits of writing component-oriented software is that the developer writes to the component standard, not the operating system. It is the component standard that interacts with the operating system to resolve the low-level requirements. Therefore, if Microsoft can move developers onto using their component standard, they can further expand regardless of operating system, and therefore hardware platform. As DCOM is also a published standard, they can also rely on other vendors to develop ports for different platforms (for example, Software-AG).

We can view the CORBA faction as having a lot of drive from the desire to compete with Microsoft. While CORBA originates from the need for a world-wide standard for distributed object communication, the competition against Microsoft seems to have caused a deviation from the original vision, resulting in a group which reacts to Microsoft's directions, rather than pushing forward with independent technologies. This is illustrated by the following quote:

Microsoft is just a company, not a force of nature. Its not the biggest company in the world, not the richest, not even the biggest seller of packaged software (that's IBM). We reel at the mere thought, but Microsoft can be dislodged from its place at the centre of the software universe. How? [163]

Consider also the following from the CORBA faction (taken from a press release regarding the Enterprise JavaBeans specification):

Theories are circulating about the merger of key specifications for application components, which would help prevent Microsoft taking control of the object development market...

... Keith Jaeger, head of international product development at tool company Synon, said he expected CORBA and EJB to merge as early as the end of 1998, to prevent the industry splintering into two camps and handing the market to Microsoft. [39]

While EJB originated from the need to provide an effective distribution standard with the JavaBeans component model (merging with CORBA), the published specification seems to draw greatly from the MTS model for component development, both essentially being ways to develop component-based, transaction-oriented applications. A comparison of the two [137], demonstrates these similarities. Admittedly, as a Microsoft white paper, the demonstration is biased toward Microsoft, but it does provide a few salient points.

The majority of marketing to encourage an organisation to adopt component-oriented techniques focuses on the potential benefits: component-orientation is an effective way of achieving large-scale reuse, it provides numerous ways of easing the development process, and makes the management and maintenance of software projects more straightforward. However, we have to consider what the people who market these technologies have to gain. By controlling the dominant architecture, an organisation can bypass the battle for supremacy with operating systems and ensure that their products and technologies are used across numerous platforms. Therefore, we must conclude that market forces play a large role in developing the ways in which software is written. Even with a technology that is potentially advantageous to developers, if its vendors do not see any market potential for its use, it is unlikely that it will move into the mainstream.

2.6 The Philosophy of Component Orientation

From the above discussion, we can see a very positive view of component-orientation. However, we must also be aware that this view is drawn from industrial literature. In drawing together this discussion, we can identify a number of common beliefs regarding component-orientation. As these beliefs influence both the propositions for case study research and survey construction (see chapter 4), they are drawn together in this section as a core philosophy of component-orientation.

1. Component-orientation increases development productivity through software reuse.
2. Component-orientation enables cross-platform and cross-language interoperability
3. Component-orientation will reduce maintenance costs and increase reliability
4. Component development is made possible through component standards
5. Component-orientation provides functionality to aid in the distribution and scalability of applications

Finally, an aspect that has not be addressed in the above discussion, but one that could be considered part of conventional wisdom about component-orientation, relates to the adoption of component orientation. A common point in discussion of adoption (for example, see [33], [38], [28], [37]) is that adoption has to be comprehensive across an organisation – it is not possible to gain the benefits of component-orientation if it is used as a simple development technique without strategic considerations. Two separate issues can be drawn from discussion. Firstly, organisations should embrace a reuse culture that is enabled by component technologies. This can be seen to be influenced by the wider domain of software reuse where, it is argued, success can only result from systematic software reuse strategies embraced by the whole organisation [82]. The second issue to be drawn from this discussion is the replacing of existing development tools with

component technologies. In this case, the belief is that component-orientation has to be used as the single development technique in order for its use to be successful – it cannot be mixed with other development technologies. Therefore, two aspects of the philosophy of component-orientation are:

6. In order to be successful, component-orientation should drive an organisational reuse culture.
7. In order to be successful, component-orientation should replace existing development techniques.

2.7 Chapter Summary

While as a concept component-orientation has existed for a long time, it is only recently that some of the aspects discussed in the seminal paper on the topic are becoming realisable. The advent of component standards provides developers with common platforms on which to develop reusable software components. Component standards provide core functionality for the structuring and interoperation of software components. The two main standards are Microsoft COM and OMG CORBA. The emergence of these standards can be seen as a step toward large scale software reuse, as well as a number of forces within the software industry, they provide a foundation on which to build a component-oriented software environment.

This chapter focuses upon literature relevant to the research aims, putting forward arguments for the chosen direction in this project. The review is divided into two distinct areas - the assessment of software technologies and theories of technology adoption. These considerations strongly influence the research approach - each area provides a foundation on which to draw when considering the research approach in this project and also the development of results following experimentation. As such, work discussed in this chapter is returned to throughout the thesis.

3. Assessing and Adopting Software Technologies

As defined in section 1.2, the overall aim of the programme of research was:

- to review the nature of software development and the concept of component-orientation, and to assess the impact of component technologies upon the software development field. Research should provide evidence of their impact, and develop guidance for the future use of such techniques.

This raises the obvious questions:

1. How can we assess the affect the new technology has upon the development process?
2. How are new technologies adopted by organisations?

In this chapter, literature on both the assessment of software technologies and technology adoption is reviewed.

3.1 How Can We Assess a New Technology?

Understanding methods for the assessment of a new software technology is vital in order that component orientation is effectively examined and realistic conclusions are drawn. The process of assessment enables us to get a good understanding of the effectiveness of a new technology, but more importantly, it allows practitioners to make more informed decisions on what technologies should be adopted into mainstream software development.

However, this assessment is never a straightforward task. There are many references in the literature to the difficulty of this experimentation within the field (for example [8], [9], [107]). The term *empirical software engineering* [9] refers to the building of knowledge from observation, formulating theories and experimentation in order to understand aspects of the discipline. From this field that we can draw knowledge related to the evaluation of software techniques and technologies through theorising and experimentation. Victor Basili, one of the most widely cited researchers in the field, argues that the underlying paradigm of software engineering should draw from the empirical nature of other disciplines, such as physics and medicine [9]. He also differentiates between the roles of the researcher and the practitioner in software engineering:

The role of the researcher is to build models of and understand the nature of processes, products, and the relationship between the two in the context of the system in which they live. The practitioner's role is to build "improved" systems, using the knowledge available and to provide feedback.

This statement identifies an explicit relationship between the researcher and practitioner, and its active nature – research should not be carried out in isolation from the practitioner, and the practitioner should learn from research.

Within the same paper, Basili acknowledges that the field is in a very primitive stage of development. We should not only consider data collection and analysis, but also the method of investigation. Basili and Lanubile [8] further consider problems with experimentation in software engineering. Due to the nature of software engineering it is not possible to directly draw from other scientific disciplines – software development will not continually produce the same product for assessment, as with manufacturing. Developed software cannot therefore be measured against replicated data points to statistically test hypotheses. The technologies and theories in software engineering tend to be human based, so the variability of human performance can also affect the experimental process.

The credibility of research in software engineering is also considered, in particular the *internal*, *external* and *construct* validity of the experimentation. Internal validity refers to the causal relationships in the study, such that a condition can be seen to lead on to other conditions, rather than relationships between such effects being spurious. External validity relates to the extent to which the experiment's findings can be generalised. Finally, construct validity relates to the selection of measurements that correctly reflect the research questions. It is against such evaluation criteria that an experiment or study should be considered in order to assess the credibility of findings.

3.1.1 Defining Empirical Software Engineering

Figure 3-1, taken from [9], defines the nature of empirical software engineering and the elements that will comprise a study based on that paradigm. The important aspects to note from the figure are the relationships between the elements in the domain. In essence, the figure is relating the

software-engineering context (the world) to theory, models and research questions. A theory is generated from observing the “world” and attempting to describe a phenomenon. A model is an expression of the theory – the model will enable aspects of the theory to be tested. The research questions are used to guide the investigation of the theory by forming hypotheses to test. The nature of the hypotheses also guides the research design, the most suitable techniques being chosen based upon the hypothesised statements.

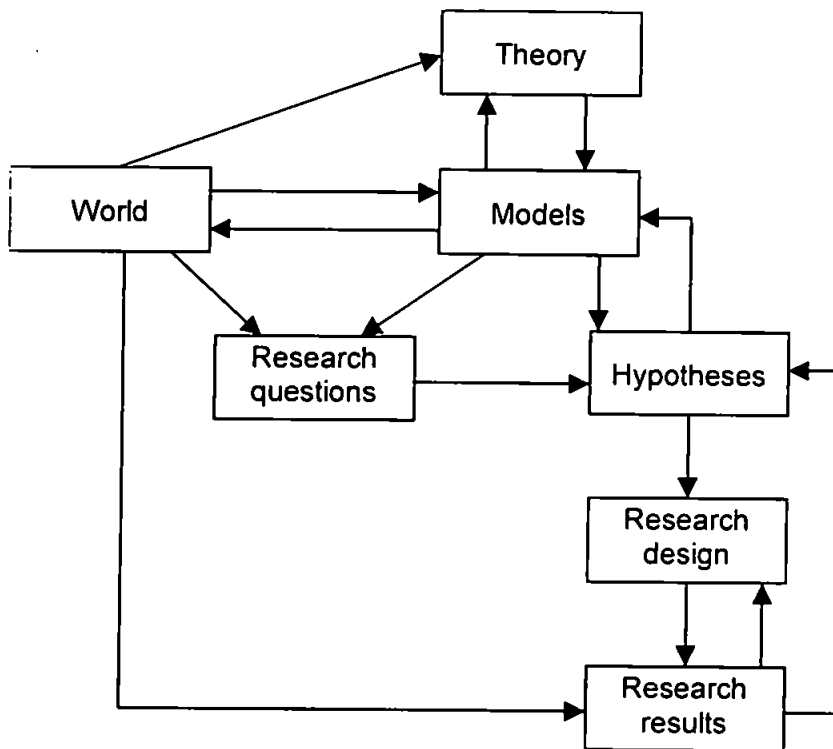


Figure 3-1 – The Relationship between ESE, models and research questions

It is acknowledged that while Figure 3-1 defines the “perfect” study, in the reality of a software engineering study, factors already mentioned (lack of data points, human factors, complexity) can affect the study’s completeness [8]. However, an incomplete research model can still be used for effective research, as long as the researchers are aware of the problems with the model, and report on these flaws in any research conclusions. Conclusions from a project should be verifiable by

others, and providing others with an awareness of potential flaws enables greater understanding of them.

A final comment drawing conclusions from an empirical study also comes from [8]. Basili and Lanubile state that drawing general conclusions is difficult due to the contextual nature of an experiment or study. This issue is important when developing the results of a research study, and is one that we will return to later in this thesis (see chapters 7 and 8).

3.1.2 Techniques for Empirical Study

One of the seminal papers for empirical software engineering came from Basili et. al. [12]. In this paper, the authors defined a framework for experimentation within software engineering. This was one of the first to propose a more rigorous structure to research in the field, moving away from the informal nature of previous experiments. The framework defined categories that applied to phases within the experimentation process (definition, planning, operation and interpretation). It aimed to formalise researcher's thinking when carrying out experimentation, so they would define both purpose and object of study before commencing assessment.

However, while that paper focused upon techniques for experimental research, numerous techniques have emerged from the field for the evaluation of software technologies. Zelkowitz & Wallace [167] examined experimental models of the validation of software technologies from a different viewpoint, grouping techniques into three broad categories:

- **Observational:** Collects relevant data from a project as it develops
- **Historical:** Collects data from projects that have already been carried out.

- **Controlled:** The classical model of experimental design from other scientific disciplines
 - multiple instances are carried out in a controlled environment to replicate and provide validity for results.

Within each category, Zelkowitz & Wallace define a number of methods, discussed below.

3.1.2.1 Observational Methods

Project monitoring: The simple process of collecting data from a project as it develops. A passive model that takes whatever data is generated by the project, it does not relate to any research questions, but can be used to establish baselines, such as those in the Quality Improvement Paradigm (see section 3.1.4.1).

Case study: A more guided approach to data collection from a live project, the acquisition strategy is guided by research questions and goals. Therefore, the data is relevant to specific research areas, which are defined before the project starts. The strength of this method is that it occurs on a live project so criticisms often levelled at software engineering research (see section 3.1.3) do not apply.

Assertion: A study where the researchers are also the practitioners. This type of study can be flawed due to bias, where the researchers can guide their practice to reflect their hypotheses. There is, however, value in researcher/practitioner approaches in a large industrial context, where the researcher does not have control over the project – in this case the research could be considered a case study.

Field study: Comparing the data collected from a number of projects simultaneously in order to try to achieve replication. A problem with this approach is that the context of each individual project may affect the data's generalisability. However, it is a good technique to try to demonstrate the replication of phenomena.

3.1.2.2 Historical methods

Literature search: Reviewing previous studies of a particular phenomenon. This approach can be used to confirm an existing hypothesis or to enhance data collected from a project by comparing it to previously published data.

Legacy data: Again, the use of previously collected data, although in this case from project documentation, rather than published findings.

Lessons learned: Often produced following an industrial project, these reflect on what occurred during the project, so that others can learn from the mistakes.

Static analysis: Researchers obtain information on a completed product. This can be likened to legacy data, but whereas legacy data examines the whole development process, static analysis focuses only on the end product.

3.1.2.3 Controlled methods

Replicated: An evaluation is carried out in an experimental setting (i.e. a laboratory rather than in an industrial project), where researchers try to replicate the experiment while changing one of the control variables (for example, changing programming language for each experiment). This controlled variation of variables enables a greater degree of statistical accuracy than is possible with case studies.

Synthetic: Due to the expensive nature of “real world” experiments, a lot of experimentation is carried out in a scaled down, “synthetic” environment. This can be beneficial to strengthen statistical accuracy within a timescale, but can be hampered by lack of industrial accuracy.

Dynamic analysis: As with static analysis, this method focuses upon the end product rather than the development method, but the product is dynamically analysed, for example, through the use of debug statements within the product code. If similar techniques are used on a number of products, comparative data can be drawn for assessment.

Simulation: Related to dynamic analysis, the end product is executed in a simulated setting in order to test its performance and behaviour. Again, this is a useful technique to gain greater statistical accuracy (reflected in the controlled nature of the execution environment). However, it also suffers due to lack of real world context.

As well as defining these different techniques, the authors also commented on the strengths and weaknesses of each approach. These are reproduced in Table 3-1:

Validation method	Strength	Weakness
Project monitoring	Provides baseline for future; inexpensive	No specific goals
Case study	Can constrain one factor at low cost ²	Poor controls for later replication
Assertion	Serves as a basis for future experiments	Insufficient validation
Field study	Inexpensive form of replication	Treatments differ across projects
Literature search	Large available database; inexpensive	Selection bias; treatments differ
Legacy data	Combines multiple studies; inexpensive	Cannot constrain factors; data limited
Lessons learned	Determine trends; inexpensive	No quantitative data; cannot constrain factors
Static analysis	Can be automated; applies to tools	Not related to development method
Replicated	Can control factors for all treatments	Very expensive; Hawthorne effect ³
Synthetic	Can control individual factors; moderate cost	Scaling up; interactions among multiple factors
Dynamic analysis	Can be automated; applies to tools	Not related to development method
Simulation	Can be automated; applies to tools; evaluation in a safe environment	Data may not represent reality; not related to development method

Table 3-1 - Methods for software engineering research

² A case study enables the examination of an aspect of software development in context without the additional expense of, for example, setting up laboratory experiments

³ The Hawthorne effect relates to the phenomenon of workers within a study carrying out their tasks with greater conscientiousness than they would in their general day to day work due to the assumption that they will be under greater management scrutiny [89]

One type of assessment of briefly touched upon in the Zelkowitz and Wallace paper, but dealt with in more detail in other literature, is feature analysis. This is a term that can be applied to a number of different techniques, all aimed at focusing upon an aspect of a software product. While Zelkowitz and Wallace's static and dynamic analysis could be likened to feature analysis, Kitchenham [89] addresses it in far greater detail in a series of articles reporting on the UK DESMET project for the evaluation of software tools and techniques. Feature analysis identifies a user requirement and maps the requirement onto features a product should possess. This is the most common method of evaluation in popular personal computer press. For example, a group of word processors will be compared against such features as ease of use, formatting capabilities, and integration with other office applications, etc.

Feature analysis is also referred to in an earlier paper by Brown & Wallnau, from the Software Engineering Institute [27], who propose a framework for evaluating software technology based upon it. The framework develops a reference model for a given domain, and then maps of a chosen technique or technology to the reference model, based upon features identified in the reference model. While this model is interesting in its use of a feature oriented view of technology evaluation, it is also interesting to note the use of reference models as a tool for understanding common aspects of a domain. We will return to the role of reference models later in the thesis.

3.1.3 Criticism of Software Engineering Research

The above reviews methods for the evaluation of software engineering tools and techniques in order to help in considering the effect of component-orientation upon software development practice. However, it should be noted that there has been criticism of software engineering

research in the past. The following section reviews such criticisms to shed light on the choice of research method for the present programme.

Criticism is often levelled at the fact that while there is a great deal of software engineering research carried out, very little seems to transfer into software practice. Sommerville [144] discussed this problem at length with regard to software process research – an area of considerable research effort but little impact upon software practice. Returning to the introductory comment at the start of section 3.1, the main goal of experimentation should be to provide practitioners with the sort of knowledge that will enable them to make informed decisions regarding the selection and adoption of new technologies.

The main criticism of software engineering research from industry practitioners comes from the lack of industry involvement. A group of publications from 1993/94 focus upon this issue. An argument from Potts [121] relates to the model of software engineering research he referred to as “research then transfer”. In this model, the researcher carried out work in isolation from industry until such time that they felt their research needed validation through practice. This process can take years of experimentation and refinement until there is some industrial involvement. The problems of transfer were further compounded due to the assumption that it would happen automatically at the end of the experiments. There was little concern with understanding how the technology could be transferred once it had been developed.

Potts argued that as software engineering practice began to mature, software engineering research should move away from “pure” scientific research, such as the continual development of new languages and techniques, toward looking at ways to understand the strengths and weaknesses of

existing approaches. A maturing industry is less willing to constantly change practices, and is more concerned with getting the best out of what they use.

This change in the pattern of research is referred by Potts as the “industry-as-laboratory” approach, where problems are identified through the close involvement of industry. The problem can then be developed in an industrial context in order to analyse, create and evaluate possible solutions. NASA’s Software Engineering Lab (SEL, see section 3.1.4.1) is identified as the pioneer in the industry-as-laboratory approach. Basili [9] goes so far as to state that the software engineering researcher’s laboratory can only exist where practitioners build software systems.

A similar argument is put forward by Fenton, Glass & Pfleeger [51] when considering the effectiveness of software engineering research and its transfer into practice. Again, they question why so little software engineering research is taken up by industry, and they question the validity of “intuitive” research, where the effectiveness of a technique is considered through the experience and analytical qualities of the researcher, not empirical evidence. They present five questions that they feel should be, but rarely are, asked about any claim arising from software engineering research:

- Is it based on empirical evaluation and data?
- Was the experiment designed carefully?
- Is it based on a toy or real situation?
- Were the measurements used appropriate to the goals of the experiment?
- Was the experiment run for a long enough time?

In agreement with Potts, they state that evaluative research must involve realistic subjects and realistic projects if it to be of use to software engineering practitioners. They too hold up the SEL approach as the “best practice” in this type of software engineering research.

A slightly different angle in approaching the same questions came from Glass [63]. His paper, extremely critical of software engineering research, took the perspective of being written in the future (2020) and examining the approaches used by software engineering research in the late 20th century. In examining approaches to research, Glass claimed that the vast majority of experimentation with new techniques was carried out using what he referred to as “advocacy research”:

1. Conceive an idea
2. Analyse the idea
3. Advocate the idea

In Glass’ vision of the future, he saw the practitioner-focussed approach that dominated software-engineering research as also coming from the SEL mode. In this future research approach, he saw three benefits:

- Software practice and research work together to carry out ideas and address problems in a practical setting
- Good research results make it into practice
- Bad research ideas are discarded quickly.

While the practice of empirical software engineering grew throughout the nineties, driven on in part from the SEL approach, some still argue that software-engineering research is still too set in

scientific principles. In a paper from 1999 on empirical software engineering, Pfleeger [120] discussed the limitations of absolutes in the evaluation of new technologies and techniques, and the problems with total reliance on measurement and experimentation as the sole means of assessment. This centres on the cause and effect assumption in some software engineering research – you do something, and an expected effect occurs. Pfleeger argues that software engineering is more stochastic in its outcomes – there is a probability distribution of an effect occurring, based upon the social and/or organisational context in which it occurs. Therefore, the goal of research should not be to determine the absolute effect in all cases, but to understand the likelihood that, under certain conditions, a particular technique will lead to improved software.

In considering methods to achieve such understanding, consideration is given to the current practice of developing a theory, building evidence to support the theory, and then replicating studies to support the theory. Two problems exist with this approach:

1. the time taken to carry out the replication
2. the changing nature of the development technique being assessed can result in a wasted study

This second point is an important one in considering the nature of software development technologies. Industry vendors generally drive newer technologies, such as object and component techniques. These vendors will continually review and refine their “products”. Therefore, in assessing such technologies, it is important to be aware of their potential volatility. A replicated study may conclusively prove a theory, but will suffer if findings come too late to be of use to industry. In addressing this problem, Pfleeger considers a sequential approach to research, comparing this to an iterative software development approach. However, whereas a developer would analyse, design, develop and iterate, the research would study, theorise, and iterate. Using an example from social sciences, the author considers research assessing the effectiveness of a

new reading technique. The technique would be defined and then tried out on a test group. The technique would be evaluated based upon the findings from the first study, refined, and iterated with another test group. In this way, the research technique would be being refined based upon interim findings.

In this and also a later co-written paper [119] the author also considers the issue of evidence in validating theories. Consideration should be made to the nature of the audience for evidence, and also its criticality, rather than assuming absolute proof is needed in all cases. Returning to the divide between researchers and practitioners, we can consider evidence prepared for the academic community to be the same as that prepared for industry. In a study carried out by Zelkowitz et. al [169], the question of the value of a research method was posed to both researchers and practitioners. The findings were that each study group had a preference for methods that reflected their own experiences – researchers preferred controlled experiments, while practitioners preferred case and field studies.

The implications are far reaching – researchers have to ask themselves why they are studying a particular technique or technology? Is it to promote further academic research, or it is to aid industry in the adoption of new techniques? If it is the latter, than the validity of controlled, laboratory experimentation should be questioned, especially if practitioners are more likely to value the results of a case study.

Therefore, in considering a study or evaluation, Pfleeger & Menezes [119] argue that it should not be just the assessed innovation that comes under scrutiny, but also the evidence produced by the study. To assess the body of evidence from a study, they define the following approach:

1. look at the innovation's previous uses

2. compare old ways with new ways
3. determine (using case studies, surveys, experiments, feature analysis and other techniques) whether the evidence is conflicting, consistent and objective.

However, a final note from this paper returns to the argument by Potts related to the assumed transfer of an innovation following study. The authors agree that an understanding of how innovations are adopted is important on top of effective evaluation. They also refer to the support infrastructure that can be provided by training materials, tools, written materials, etc. This point is considered in more detail in chapter 8.

3.1.4 Software Process Research

In determining methods for the evaluation of software technologies, consideration must also be given to software process research, in particular process improvement efforts. While the above discussion touches upon the most explicit techniques within the field for technology assessment – the work of the Software Engineering Laboratory, in particular the Quality Improvement Paradigm (QIP), the following places such methods in the context of software process research.

Sommerville [145] provides a model that defines the contributing factors to product quality. This model is illustrated in Figure 3-2. The primary concern for this research programme is the development technology aspect of the model – if suitable technologies can be used effectively to their strengths, this will greatly contribute to the quality of the end product. However, process quality is also an important factor and, as discussed below, there is a relationship between technology assessment and process quality that some process improvement efforts have attempted to address.

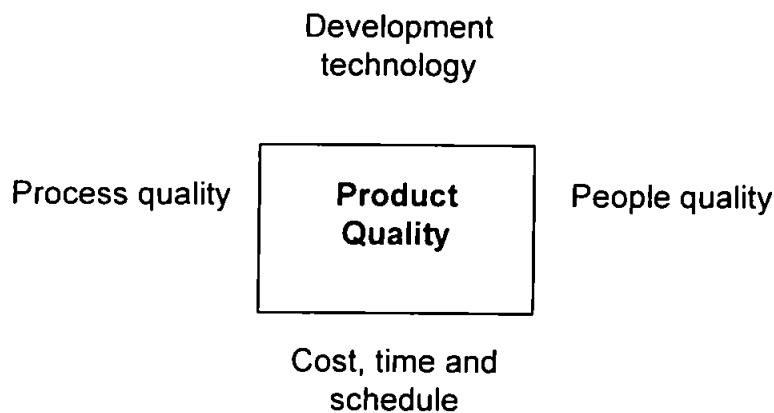


Figure 3-2 - Factors affecting product quality

The general area of software process research is, according to the Software Engineering Body of Knowledge (SWEBOK) knowledge area (KA) of Software Engineering Process [49], broken into four themes:

1. Process definition
2. Process assessment
3. Process improvement
4. Process support

Process definition and process support aim to define the software development process. Aspects such as process models and modelling techniques [45] can be covered in these themes. The KA defines a number of possible reasons for process definition and support, such as facilitating human understanding and communication, supporting process improvement and supporting process management. It states that the level of definition will depend upon this need. To illustrate this point, the level of process definition in the two case studies presented in the research

programme (see chapters 5 and 6) is high, as they are there solely to help facilitate understanding related to the assessment of component technologies within the cases. The use of the process definitions in the case studies is discussed further in section 4.2. However, if such a process definition were to be used as the foundation for a process improvement programme within an organisation, such a definition would be far more detailed.

It is the areas of process assessment and improvement that are our main focus for this examination, as this is the domain in which the SEL work lies. While the general term for such approaches is generally referred to wholly as software process improvement, there are aspects of both assessment and improvement within such an approach. Figure 3-3 illustrates the “process improvement process”, as defined by Sommerville [145].

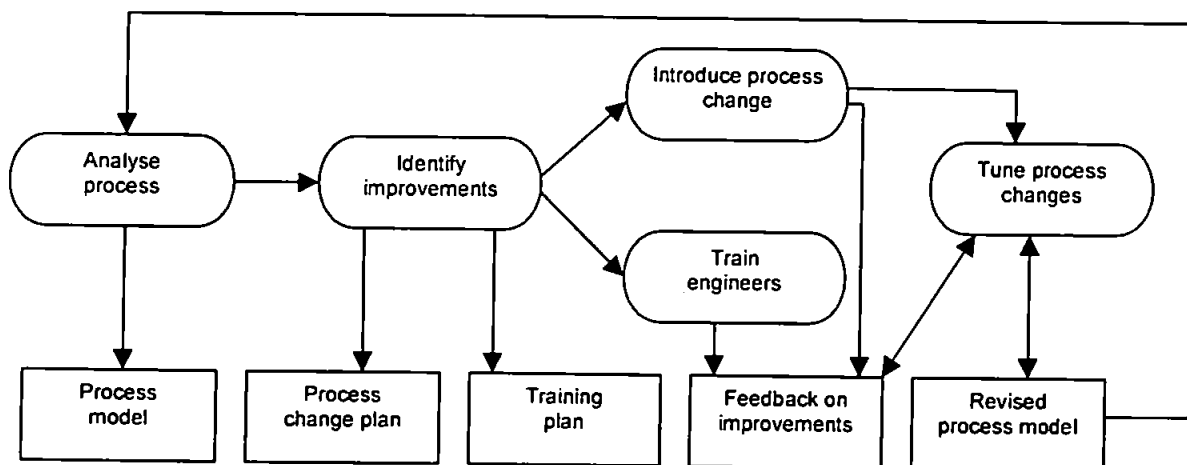


Figure 3-3 – The Process Improvement Process

This reflects the Software Engineering Institute’s (SEI) Initiating, Diagnosing, Establish, Acting and Leveraging (IDEAL) model [99], which is intended to provide guidance for organisations in the planning and implementation of a process improvement programme. Figure 3-4, taken from [141] illustrates the IDEAL model.

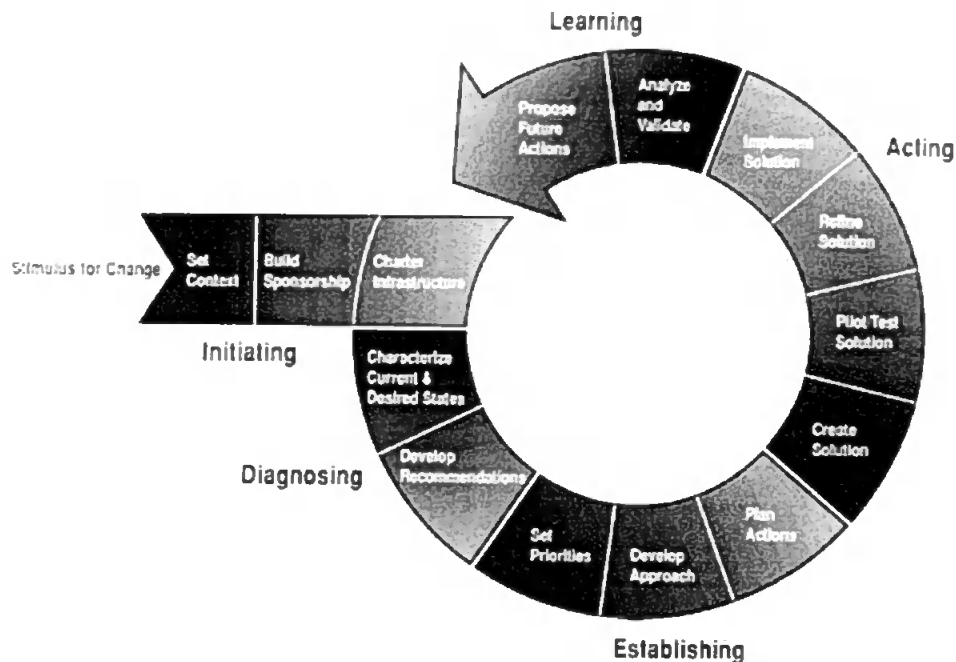


Figure 3-4 – The IDEAL Model

Both the model described by Sommerville and also the IDEAL model demonstrate the cyclical nature of the process improvement process, and also a distinction between methods of process assessment before improvement can be carried out.

The most well known approach to software process assessment comes from the SEI's Capability Maturity Model (CMM) [73]. The model provides a process assessment that enables the classification of a software process into one of five different levels:

1. **Initial** – an organisation that does not have any effective management procedures or project plans and no formal process model.
2. **Repeatable** – an organisation does have management procedures and project plans, but no formal process model.

3. **Defined** – an organisation has management procedures and project plans, plus a process model that can be used as the foundation for process improvement
4. **Managed** – As with level 3, but the organisation also has an effective strategy for the collection of quantitative data to feed into process improvements
5. **Optimizing** – An organisation that is committed to constant process improvement. Process improvement is an integral part of organisational strategy.

The definition of levels of assessment was extended in a revision to the CMM [116] to include key process areas, which an organisation would have to have in place to be assessed at a given level.

While the CMM is the dominant process assessment technique in the USA, a European funded project called Bootstrap [90] attempted to develop a similar approach to European software practices. Bootstrap developed upon the assessment approach of the CMM, taking a similar approach to the CMM, but also integrating some issues from the ISO 9000-3 [77] to develop the method for assessment and classification of an organisation's software practices. The Bootstrap method defines a reference framework that describes typical software development practices that is used as a model for comparison when assessing an organisation's practices.

Both the CMM and Bootstrap have an underlying commitment of enabling the effective introduction of new technologies, the belief being that without effective methods and practice, effective technology adoption is not possible. Throughout the CMM level, reference is made to expected levels of technology measurement, from ad hoc data collection at level 1, through qualitative data collection and sharing at level 3 to quantitative assessment, proactive evaluation and use in process improvement programmes at level 5. A level 5 key process area, defined in

CMM 1.1, is technology change management. Similarly, the Bootstrap method defines, as part of its reference model for typical development practices, a technology grouping [150]. This technology grouping defines the need for technology management – specifically methods for evaluating the relevance of new technologies, supporting the introduction and placement of the technologies, and managing technology integration⁴.

While such models are sometimes referred to as process improvement methods, what they actually provide is model of what the software process for an organisation should look like, and assess an actual process based upon this model [24]. The improvement aspects for the method are left to the discretion of the organisation. Two explicit approaches to software process improvement are discussed below. Additionally, there has been work in applying the CMM approach to process improvement [117] that provides a set of guidelines addressing the level 5 key process area of technology change management. The approach advocates the establishing of a technology management group, whose role is to introduce and evaluate new technologies and to manage technology change. It encourages an aggressive approach to the identification of new technologies and the piloting of evaluative projects to assess them. This aspect of process improvement very much addresses an approach to the evaluation of software technologies that is related to an organisational approach to process improvement. Its evaluative approach can be compared in some ways to the QIP approach (see section 3.1.4.1) in that it focuses upon the

⁴ From the SEI viewpoint, a different initiative is in place for the transfer of technologies, the Transition Enabling Program, that defines a *transition package* [57] that provides organisations with a knowledge base for the use of a given technology. This is a technique we shall return to when considering the development of research results.

quantitative measurement of technologies, and uses this measurement to aid in the adoption process.

Assessment approaches discussed above have contributed toward an international effort to define a set of standards for process improvement – the ISO SPICE project [47]. The SPICE (Software Process Improvement and Capability Assessment) project had three principle goals [148]:

- to develop a working draft for a standard for software process assessment;
- to conduct industry trials of the emerging standard;
- to promote the technology transfer of software process assessment into the industry world-wide.

The outcome of this project has been the development of ISO/IEC 15504, a multi-part standard for process assessment and improvement. These standards can be divided into two distinct areas – those that are prescriptive (or normative) which define practice that must be adhered to for an organisation to claim it is ISO/IEC 15504 compliant, and informative aspects, which provides guidance for certain aspects of process improvement. Normative aspects relate to:

- a reference model for processes and process capability;
- performing process assessment;

whereas informative aspects are:

- concepts and introductory guide;
- guide to performing assessment;
- an assessment model and indicator guidance;
- guidelines to qualification of assessors;

- guide for use in process improvement;
- guide for use in determining supplier process capability;
- vocabulary.

When considered against our own requirements in examining approaches to the assessment and transfer of new technologies, the SPICE standards do not provide explicit detail for a specific approach. While the management of technology is discussed throughout various aspects of the standards, no explicit reference is made to techniques for technology assessment. In particular, this is evident in the guidelines for technology assessment when using SPICE for process improvement. There is nothing that is directly comparable to the key process area of technology management from Paulk et. al.'s [117] approach with the CMM - the informative guide to process improvement using SPICE [147] makes no explicit reference to technology assessment.

Finally, in our review of software process assessment and improvement approaches, we return to the Software Engineering Laboratory (SEL) method. This approach to process improvement defines a specific method, and provides a view of technology assessment and adoption. We have already briefly considered this approach as it is often referred to in software engineering research literature. The following discusses the approach in more detail.

3.1.4.1 The Software Engineering Laboratory Approach to Evaluation and Improvement

As can be seen from the above discussion, there are differing viewpoints in assessing software technologies. The research viewpoint focuses upon techniques for assessment and models for research. The process-oriented view focuses initially upon software development practice and considers technology assessment and transfer to be possible only in a controlled software development process. The SEL approach, however, is consistently held up as a good model for

software engineering research, and also held in high esteem within the software process community. While as a whole it appears to be a process assessment and improvement model, it defines techniques for both the evaluation of technologies and also the communication of experience related to the technology. As such, it is something that we will return to throughout this thesis. In this section its role as a technique for technology assessment is considered.

The SEL model is often referred to as the Quality Improvement Paradigm (QIP). It is a vehicle for continuous development process improvement using iterative assessment and transfer of both process and development technologies based upon measurable experiences with the new techniques.

The phases of the Quality Improvement Paradigm (QIP) are defined in [7] as:

1. Characterise the project and its environment
2. Set quantifiable goals for successful project performance and improvement
3. Choose the appropriate process models, supporting methods, and tools for the project
4. Execute the processes, construct the products, collection and validate the prescribed data, and analyse the data to provide real time feedback for corrective action.
5. Analyse the data to evaluate current practices, determine problems, record findings, and make recommendations for future process improvements.
6. Package the experience in the form of updates and refined models, and save the knowledge gained from this and earlier projects in an experience base for future projects.

The SEL approach has been applied to both process techniques such as the Cleanroom methodology, and also to development techniques such as object-orientation (for example, see [10] and [168]). In terms of the evaluation of a software technology we focus upon phases 1-4 of

the model. The packaging and sharing of experience (the Experience Factory [11]) is of concern in considering the adoption of a technology, and is addressed later in this thesis (see chapter 8).

The evaluation of a given technique or technology hinges on the quantitative evaluation of a product based upon clearly defined measures. In determining the parameters for this evaluation, a Goal/Question/Metric approach is used. This is defined in [7] as a measurement model on three levels:

- **Conceptual level (goal):** Goals are defined for a technique or technology (termed an object), relative to the environment in which it is to operate.
- **Operational level (question):** A number of questions related to the object are defined in order to achieve a specific goal.
- **Quantitative level (metric):** A set of metrics, associated with questions in order to answer them in a quantitative way. This, in turn, relates quantitative measure to the goals of the assessment of the object.

Through the definition of these values, the experimenter has measurable goals for the assessment of the object.

As an assessment technique for software technologies, the major benefit of the QIP approach is that it is carried out within software practice. It provides a model for the collection of data from a live project that enables the evaluation of a new technique without imposing on the project execution. It can be argued that it is this focus upon the practitioner environment that has resulted in so much positive evaluation of the SEL approach.

3.2 Adopting Software Technologies

The following section addresses the question:

How does an organisation adopt a new software technology?

In doing so, the first consideration has to be why there needs to be an understanding of the process of adoption within an organisation. Returning the Potts' critique of software engineering research [121], he refers to the nature of the transfer of a lot of research findings into practice – there is an assumption that research will transfer with no consideration of how this will be done. The experimentation and results form only the initial evaluation of, for example, a software technology – it aims to determine the benefits and potential problems with its use. The research can only be valuable if it helps an organisation's decision to adopt a technology, and, once undertaken, how it is used.

From the viewpoint of the research programme, the case study approach results in theories of the use of component technologies within the development process. These theories are tested against a practitioner survey to identify common problems with the adoption and use of these technologies. The resulting material provides the foundation to a body of experience in the use of component technologies that can be shared with practitioners in order that they have a greater awareness of possible problems that could affect their component-oriented development effort. Therefore, it is important to consider how to best present that experience to the practitioner. By examining theories of technology adoption, the aim is to understand how an organisation goes about adopting new technologies, and how it develops organisational knowledge regarding them.

While the majority of literature focuses upon Diffusion of Innovations (in particular) and also Network Effects, recent work also looks at the flaws in these approaches and the contribution Organisational Learning can play in ensuring an effective adoption of a new technology [18, 5].

The following addresses these classical models and considers their suitability to the needs of this research project.

3.2.1 Diffusion of Innovations

The theory for the Diffusion of Innovations comes from work by Everett Rogers [133]. Rogers proposed that the way in which a new technology is adopted into the mainstream follows a model influenced by understanding, information, communication and social structure. This model has four main elements:

1. Innovation
2. Communication
3. Adoption
4. Social system

3.2.1.1 Innovation

An innovation is an idea, practice or object that is to be perceived to be new by the innovation consumers. It has a number of characteristics that can affect its adoption:

- **Relative advantage:** the degree to which an innovation improves on the idea it supercedes.
- **Compatibility:** the degree to which an innovation is perceived to be consistent with existing practice, values or needs.
- **Complexity:** the degree to which it is considered difficult to understand and use.
- **Trialability:** the degree to which it can be experimented with in order to assess its effectiveness.
- **Observability:** the degree to which the results of the innovation are visible to others.

3.2.1.2 Communication

The communication process between innovators and adopters is crucial in the diffusion of an innovation, as it is via communication channels that knowledge can be transferred, putting adopters in a position to make an informed decision regarding the innovation. Rogers defines two kinds of information within the communication process:

- **Hard information:** information relating directly to the innovation, such as how it works, how it should be used, etc.
- **Soft information:** information relating to the innovation's cost, potential benefits, evaluation factors, etc.

While the hard information provides adopters with facts relating to the technology it is primarily soft information that influences the adoption decision.

The communication channels themselves also have different effects upon the diffusion of innovation. Again, two types of communication channel are defined:

- **Mass media** – such as television, radio and advertising, communicating initial information about the innovation to many potential adopters
- **Interpersonal** – which aid in the persuasion of the adopter to take up a technology arise between adopters and consultants, vendors, etc.

A final factor that affects the communication process is what Rogers calls the “nature of fit” between innovators and adopters. Put more simply – do the innovators and adopters speak the same language? This can be essential, as understanding is crucial in the communication of information regarding the innovation. In a homophillious relationship, where innovators and

adopters share commonality in beliefs, education, social status, technical expertise, etc., the likelihood of adoption is higher than in a heterophillious relationship.

3.2.1.3 Adoption

The adoption process can be divided into five activities:

- **Knowledge:** the activity of obtaining information about the innovation, related closely with the communication process, in particular mass media communication and hard information.
- **Persuasion:** where the diffuser attempts to influence the decision of the adopter, based again on the communication process, but involving a greater degree of interpersonal communication and soft information.
- **Decision:** obtaining evaluative (soft) information regarding the innovation, and the first time the intention to accept or reject the information may emerge.
- **Implementation:** if the adopter decided to adopt the innovation, the activity of implementing and testing the innovation in the adopter's own environment.
- **Confirmation:** a decision to accept and further integrate, or to reject, the innovation based upon the implementation.

The adoption process can depend upon the nature of the adopter. Rogers defines a bell curve of adoption based upon the type of adopter. Figure 3-5, taken from [133], illustrates this curve.

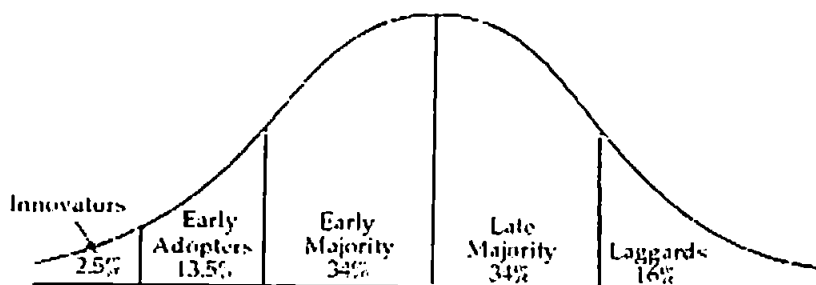


Figure 3-5 – Level of adoption based upon type of adopter

- **Innovators:** Adventurous, networked with other innovators, understanding complex technical knowledge, the ability to cope with uncertainty
- **Early adopters:** Respected within the industry, with strong opinion leaders (see below)
- **Early majority:** Long period of deliberation before adoption decision, interaction with peers, seldom hold positions of opinion leaders
- **Late majority:** adoption may result from economic or social necessity due to the diffusion effect
- **Laggards:** point of reference is the past, suspicious of opinion leaders and change agents (see below), few resources.

3.2.1.4 Social System

The social system is the environment in which the diffusion process takes place. Rogers defines 5 aspects of the social system that can affect the diffusion process:

- **Social structure:** how individuals within the social system communicate. This affects the way that information is communicated through the system.
- **Social norms:** the behaviour patterns for systems members. Social norms define the boundaries of acceptable behaviour within a social system. Rigid norms may hinder innovation diffusion, as people may not feel able to comment on the validity of an innovation.
- **Change agents and opinion leaders:**
 - **Change agent:** a proactive individual who influences innovation decisions.
 - **Opinion leader:** A respected, innovative member of the social system who influences other people's opinions.

These are critical roles in diffusion as their attitudes can greatly affect the adoption of an innovation

- **Adoption decisions:** The way decisions regarding adoption are carried out. It may occur at an individual, group, or authority level, and can profoundly affect the nature of the adoption:
 - **Individual (or optional-adoption):** a single person decides, independently of others within the social system. However, the individual may be influenced by social norms. Individual decisions allow for a rapid decision process, but can cause group resentment.
 - **Group (or collective-adoption):** a consensus of group opinion is used to decide on the adoption. This is the healthiest approach for the social system as everyone whom the adoption will affect is involved. Participant involvement in change is an essential aspect of change management [4]. This approach also allows for the most rapid diffusion, as it is likely to have the least resistance
 - **Authority (or authority-adoption):** A few key people make the decision regarding the adoption. This may or may not be fast, depending upon the level of acceptance among others within the social system.

3.2.1.5 The Influence of Innovation Diffusion upon Software Engineering

While a lot of literature refers to Rogers' model in passing (for example Fowler & Patrick [56] and Zelkowitz [168]) very few provide a detailed consideration the implications of the model on software engineering technology transfer. Raghavan & Chand [125] in their summary of earlier work, identify nine key points:

1. Diffusion is a process by which an innovation is communicated through certain channels, over time, and among members of a social system
2. The perceived attributes of an innovation have strong implications for the success or failure of its diffusion
3. Diffusion is accompanied by change, so effective change management is critical for successful diffusion
4. Diffusion occurs in a social context, so factors like social structure, culture and norms can facilitate or impede diffusion.
5. Diffusion requires effective communication, so the selection of communication channels and the match between participants are important factors in promoting diffusion
6. Innovation adoption decisions are influenced by both rational and irrational factors
7. People differ in their propensity to adopt innovations. Based on the propensity, one can group people in categories (early majority, late majority, etc.). These categories also reflect the relative order in which these people will adopt innovations
8. Innovation adoption decision processes may be carried out individual, collectively, or by authorities. The level at which the adoption decisions are made have significant implications for diffusion
9. Change agents and opinion leaders acts as catalysts during diffusion. Their attitudes toward the innovation can largely determine the success or failure of the diffusion.

They suggest that the diffusion of innovations approach is relevant to software engineering technology transfer two ways:

- **As a Descriptive Model** - to be used as a theoretical foundation for conducting empirical studies to enhance understanding of software engineering innovations, addressing questions such as:

- Are software engineering innovations diffusing as fast as they can?
- If not, what are the key problems slowing the diffusion down?

- **As a Prescriptive Model** - By refining general guidelines from Rogers' model in the context of software engineering. They identified a number of problems within the software-engineering context that would need to be addressed if the model were to be effective:
 - The abstract nature of software engineering innovations means that they are prone to misunderstanding
 - There is a need for the active involvement of researchers and innovators in the adoption process – implying a need for greater collaboration between software engineering researchers and practitioners
 - A need to be able to deal with the complexity of the social system – whereas other domains have literature relating to their management, there is no such body of knowledge within software engineering. Additionally, most software engineering professionals rarely have management training, which compounds the problems of control and change within their social system
 - The availability of information for software engineering innovations – while there is generally hard information available relating to facts about the innovation, there is little soft information that could aid the diffusion process.

To overcome these obstacles, they suggest the following:

- The adaptation of innovations diffusion literature to the software engineering domain
- The software engineering research community should become more involved in communicating innovation to practitioners

- Software engineering education should look at introducing both management and diffusion techniques into the development of new software engineering professionals.

It is interesting to note, while this paper was published in 1989, there is little evidence of these suggestions being addressed. Pfleeger & Menezes [119], highlighted this once more in a far later publication.

A paper from a business studies perspective came from Fichman & Kemerer in 1993 [52]. In this paper the authors examine the nature of software engineering innovations and produced a model to predict take up. The model used diffusion of innovations theory as a foundation, but criticised this view for considering adoption only from an organisational viewpoint. They argued that the view of the wider community also plays a part in the acceptance or rejection of an innovation. They stated that the community viewpoint is crucial in software engineering as benefits of adoption depend upon the number of current and future adopters (i.e. a technology will not be adopted by an organisation if others are not also adopting).

Therefore, their work looked at the economics of adoption – specifically, the economics of technology standards –focusing upon *increasing returns on adoption*. Increasing returns on adoption states that the benefits of adoption depend upon the size of the community of other (past, present and future) adopters.

The authors identified three issues that were particularly applicable to software engineering:

- **Learning by using** – benefits increase as experience grows
- **Network externalities** – immediate benefits of use are a direct function of the number of current adopters (see section 3.2.2)

- **Technology interrelatedness** – a large base of compatible products is needed to make the technology worthwhile as a whole.

In developing these ideas, the concept of a *critical mass* is introduced. This critical mass is needed so that the technology can achieve mainstream acceptance. If this critical mass is not achieved, the wider community will not adopt the technology. Economists have identified four factors that affect the potential to achieve critical mass:

- **Prior technology drag** – a significant installed base of prior technology can hamper acceptance of innovations
- **Irreversibility of investments** – if the adoption of the technology requires substantial investment in training, products, etc. acceptance can be adversely affected
- **Sponsorship** – an entity (person, organisation or consortium) existing to guide the development of the technology can increase adopter confidence and aid adoption.
- **Expectations** – the expectation that the innovation will be adopted can also aid in its adoption.

The conclusion we can draw from this paper is that the diffusion of innovations model may not provide the complete solution for software engineering innovations adoption, but it certainly provides a good foundation on which to consider models for adoption.

The very recent paper of Pfleeger & Menezes's [119], again looked at the influence of the diffusion of innovations model. It also stated that following Raghavan & Chand's paper of 1989, there had been little consideration of innovation in software engineering literature. The focus of this paper was different in that it attempted to determine the types of evidence needed to be

produced by software engineering innovators in order to convince practitioners to adopt their innovations. This evidence can be analogised to the soft information referred to in Rogers' model. This is the information used by the adopters to aid the decision process.

The impetus for this understanding came from earlier work in software technology diffusion by Redwine & Riddle [127], which suggested that a new technology needs to mature for 15 to 20 years before it is stable enough to be used by the mainstream. This maturation is not reflected in modern software engineering practice. For example, the Java language is only a few years old, but has been adopted by a considerable proportion of software engineering practitioners. Therefore, there is a need to understand the ways in which new software engineering technologies are presented to and adopted by practitioners, even if they cannot be considered mature. Like Raghavan & Chand [125], the authors wanted to consider work in other domains that could help in this understanding. Rogers' model for the diffusion of innovations provided the necessary foundation. The thrust of this argument is that research provides evidence to aid in the diffusion of an innovation. Conclusions drawn state that while it is difficult to understand how different technologies suit different situations, learning work in other areas enables the building of knowledge understanding the nature of innovation diffusion. This should influence how innovators and researchers present evidence relating to the innovation, enabling practitioners to make more confident decisions regarding adoption.

3.2.2 Network Externalities

While the concept of network externalities is touched upon in the paper by Fichman & Kemerer [52], it is worth further investigating this economic theory as it is often attributed to high technology markets (for example, Katz & Shapiro [86] and Bhattercherjee & Gerlach[18]). Katz and Shapiro [86], two leading authors in the area, examined the influence of network externalities

upon technology adoption. In the paper the authors analyse technology adoption in industries where network externalities are significant. The authors define network externalities as the benefit that a customer can derive from the use of goods based upon the number of other consumers purchasing compatible items. Put another way, the size of the network of consumers of similar and compatible products is in proportion to the benefits of ownership on the product. The classic network externality example of the telephone is used to illustrate this point – the more people that own a telephone, the more valuable it becomes to a given owner. If only a few people in the world owned a telephone, it would be of little value because there would be only limited communication potential. However, as a huge proportion of the world's population owns the telephone, it is extremely valuable as a communications device.

Additionally, the role of standards in relation to network externalities is discussed. The presence of standards can help to extend a network, and therefore can add value to an item, due to the compatibility that a standard provides. The example the authors give is that of PC hardware, where common interfaces, defined as standards, allow any number of hardware manufacturers to produce peripheral devices that they know will work with any PC. This is an important point to consider with the take-up of component-orientation, whose theory has existed for almost thirty years (coming from McIllroy's early paper [100]), but where industry interest was not forthcoming until standards were emerging (as discussed in section 2.5 and also identified by Chappell [37]).

A more recent article that helps in our understanding of why people adopt certain technologies comes from Liebowitz and Margolis [95], in which they further define network externalities, or *network effects* as they prefer. They refer to the phenomenon as network effects because, they argue, it is only external to the network if market participants fail to internalise these effects upon

the network. Put another way, the effects are only external until they are influencing the product network. The authors again discuss the influence of network effects on high technology industry, arguing that they undoubtedly have a great effect upon the adoption of such products and discuss the role of standards in enhancing network effects. They identify two types of network effect:

- **Direct:** the effect is directly related to the number of users of a product
- **Indirect:** the effect is “market mediated” – complementary goods are more readily available so the adoption of the product is more desirable

While the paper acknowledges the influence of network effects upon the adoption on new products, it is also careful to point out the possibility of too much reliance upon them. A number of restrictions are discussed, but one that is most relevant to this research programme is that network effects assume a homogeneous market place – where only a single technology can prevail. This is obviously not always the case – there are many instances where technologies, that could be considered incompatible, co-exist as each suits a subset of users. The classic example here is that of PCs and Apple Macs – while the PC market is stronger, there is still a market for Macs. While the two could be considered part of the same network, they manage to co-exist within it.

This point also highlights another assumption in the theory of network effects – that all consumers have the same compatibility needs. That is, all users within a network wish to interoperate with everyone else. In reality, compatibility may be required on a much smaller scale – perhaps even on an organisational or inter-personal level. Certainly compatibility is not required outside of vertical industries and supply chains. In these cases, the network effects are nowhere near as influential as one might first assume. Liebowitz and Margolis illustrate this in

relation to the Microsoft antitrust lawsuit, in which network effects were cited as an argument to demonstrate the anti-competitive practices of the company. This issue is developed in the paper, but also in far greater detail in [96]. They conclude that when considering network effects as the sole theory, one can certainly view Microsoft's practices as anti-competitive, but when one considered the variety of user needs, and also the possibility of heterogeneity within the market place, the argument is not as strong.

In further developing the concepts of network effects, Liebowitz and Margolis raise an issue that is very important to this research programme, and one which will be returned to in far greater detail in chapter 8. In considering the influence of network effects upon an emerging or developing technology, the importance of effects comes not only from ownership of the product, but from the body of knowledge about it. A large network of owners is not of itself sufficient to influence the adoption choice, as, without the backup of a body of knowledge and experience, ownership is useless because the potential of the technology cannot be exploited. As network experience and confidence grows, the expected payoff for a new adopter becomes higher, and the greater the likelihood of adoption.

Considered as a whole, these influential papers raise some important issues:

1. Direct and indirect network effects have influence upon the likelihood of a technology being adopted
2. Standards further strengthen network effects by enhancing the potential for compatibility
3. Emerging technological adoption cannot be explained solely with effects relating to ownership. A body of knowledge and network experience are important factors in influencing adoptions.

However, the work of Liebowitz and Margolis warns against overreliance on a single theory. Further flaws in both the theories of network effects, and also the diffusion of innovation are discussed in more detail below, where we consider the influence of organisational learning upon technology adoption.

3.2.3 Organisational Learning

Organisational learning is grounded in the psychological theory of how organisations, as a collective, learn. Bhattecherjee & Gerlach [18] define it as:

The process of acquiring knowledge, expertise and insights about complex innovations, and institutionalizing this wisdom by modifying organizational roles, processes, structures, routines, strategies, technologies, beliefs and values as needed.

In their review of the literature on organisational learning, Landes, Schneider & Houdek [93] stated that common principles were the capturing, storing and reusing of experiences or knowledge within an organisation. It is particularly useful when considering techniques for technology adoption as it helps us understand not only the adoption process but also how the organisation's knowledge grows following the adoption. It also tends to be critical of business/economic theories of adoption in that these only consider how the technology gets into the organisation, not what the organisation can do with it once it has been adopted.

Attewell [5] is particularly critical of business approaches, in particular diffusion of innovations. The focus of the criticism is that they mistakenly focus upon influence and communication as the main drivers for technology adoption, not knowledge and understanding. Other criticisms levelled at these approaches include:

- The emphasis upon the demand for an innovation, assuming that everyone has the same abilities and opportunities to adopt
- The number of assumptions made by diffusion theory (smooth take up, rationalistic adoption process⁵). He argues that in the case of complex new technologies, take up is never linear, and can occur at multiple levels.

In reviewing previous organisational learning literature, Attewell discusses the relevance of context, and its influence upon adoption choice. Drawing from work by Eveland & Tornatzky [50], he enumerates five elements of context:

1. The nature of the technology
2. User characteristics
3. Deployer characteristics
4. Boundaries within and between deployers and users
5. Characteristics of communications and transaction mechanisms

In the case of high technology, diffusion is more difficult if:

1. The scientific base is complex or abstract
2. The technology is fragile (in the sense that it does not work consistently)
3. It requires handholding (aid & advice) to adopters
4. It is 'lumpy' (affects large sections of the organisation)

⁵ The adoption process itself is not complex – once the company has selected the innovation its adoption will be straightforward and successful. The criticism being that diffusion of innovations only considers adoption to the point where an organisation decides to use an innovation, not its evolution through the organisation.

5. It is not easily productised

All of these aspects can be relevant to leading edge development technologies, such as, component-orientation. Similarly to the discussion above, comparing component-orientation to Kemermer & Fichman's models for adoption, we can consider the technology against these criteria:

1. Component-orientation is certainly complex, requiring an understanding of new concepts and the application of concept to technology implementation (chapter 7 illustrates practitioner opinion regarding this complexity).
2. Current implementations (of both CORBA and DCOM) are often criticised as not being complete in their representation of the standard (for example, see [132] and chapters 5 and 6)
3. The need for the sharing of experience in the use of component technologies is something to which we return later in this thesis (see chapters 7 and 8).
4. Another common point of discussion among industry literature related to component-orientation is that it is not something that can be adopted independently of wider organisational considerations
5. Referring back to point 2, component products (standards implementation, services, etc.) are complex and expensive.

Returning to the reliance on information transfer in classical diffusion theories, Attewell distinguishes between the types of information communicated between deployer and adopter in the adoption process:

- **Signalling** – communication about the existence and potential benefits of a technology. Using mass-media communication technologies this information is easily transferred.

- **Know-how / knowledge** - learning information or the communication and development of knowledge regarding the technology. This information places a far greater demand upon both the deployer and adopter.

While this differentiation of information can be compared to Rogers' hard and soft information [133], the distinction between knowledge transfer and development is important within organisational learning, whereas diffusion of innovations deals solely with knowledge transfer. The traditional view centres on the transfer of knowledge from a "knowledge supplier" (for example a vendor, a research organisation, a university/industry link, etc.) to the adopting organisation. This view does not consider how the knowledge is propagated throughout the organisation once it has been transferred. Taken to extremities, it would mean that once the adopter has acquired a few relevant papers, they will be able to exploit the potential of the new technology. In essence, the communication of knowledge is being reduced to signalling.

Attewell identifies studies demonstrating that, in reality, the knowledge required to exploit the potential of a new technology has to be developed within the organisation. This is a far slower process based on the development of experience in the technology's use. Additionally, in order to build effective knowledge regarding a technology, they need to build upon existing knowledge. This goes against the assumption identified above that any adopter has the same potential for adoption.

The spreading of knowledge within the organisation comes from individual learning that is then propagated throughout the organisation to become institutionalised. It is only with this institutionalisation that knowledge can be considered part of the organisational memory. This

process is described in the work by Crossan, Lane and White [43] on the development of a framework for organisational learning, which will be returned to later in this section.

Bhattercherjee & Gerlach [18] also criticise traditional diffusion theories. Their paper is of particular interest to the research programme as it considers the application of organisational learning to the adoption of object-orientated technologies (OOT). In considering the adoption of OOT, the authors state that

...a company's adoption behaviour derives from its ability to understand and use OOT.

The important issue here is the understanding of a technology before being able to successfully adopt it. The authors argue that an effective model of adoption derives from both external and internal influences, and also the organisation's ability to learn from these influences. They argue that, with a complex technology such as OOT, there are increasing returns as knowledge and experience both internally and externally develops. The question remains how internal knowledge can be institutionalised and whether external experiences and knowledge can be institutionalised. The authors see organisational learning as a means to achieve these goals.

Crossan, Lane & White [43] address the same question in the development of their *4I framework* for organisational learning. While the focus of their work differs from the needs identified by Bhattercherjee & Gerlach, and also this research programme, the framework provides a very good foundation in considering the adoption of an emerging technology. For this reason, it is described in some detail below, and is returned to later in the thesis (see chapter 8).

3.2.3.1 The 4I Framework

The focus of work for the 4I framework is concerned with the phenomenon of strategic renewal, the underlying aim of an organisation being continuously learning. It is based upon 4 key premises, and a core proposition:

- **Premise 1:** Organisational learning involves a tension between assimilating new learning (exploration) and using what has been learned (exploitation)
 - **Premise 2:** Organisational learning is multi level: individual, group and organisational
 - **Premise 3:** The three levels of organisational learning are linked to the social and psychological processes of intuiting, interpreting, integrating and institutionalising
 - **Premise 4:** Cognition affects action (and vice versa).
-
- **Proposition:** The 4I's are related in feed-forward and feedback processes across the levels.

There are two points that require elaboration from this definition. Firstly, it is in keeping with what has already been discussed regarding the development of knowledge within an organisation. It also refers to the construction of knowledge from previous experience. The statement "Cognition affects action (and vice versa)" refers to the influence of doing upon the understanding of a concept. Understanding guides what will be carried out, but also what is carried out will inform what needs to be learned. The concept of "learning by doing" has been discussed above and is also focus of constructivist learning [115].

The core proposition speaks of feeding forward and feeding back within the learning process. The feeding forward of knowledge is obvious in the transference of learning from individuals to groups to organisations. Feeding back, however, relates to how the changes in organisational practice (resulting from the institutionalisation of knowledge) affect people at a group and individual level.

Table 3-2, a complete version of which can be seen in [43], defines the 4I framework across the levels of the organisation. It identifies the four processes involved in the organisational learning, and the levels at which they occur.

Level	Process
Individual	Intuiting
	Interpreting
Group	Integrating
Organisation	Institutionalising

Table 3-2 - Processes in organisational learning and the levels at which they occur

Intuiting is a preconscious recognition of a pattern and/or possibilities inherent in a personal stream of experience. In other words, intuition is based upon the individual being able to relate a new concept to previous experience, and from that experience, be able to derive some form of recognition from the new concept. Therefore, an individual can get a “feel” for something they have not directly experienced, but they can relate to.

Interpreting develops the intuitive elements of the concept into a more conscious understanding. This progression develops the individual's cognitive maps⁶. This interpretation takes place within a domain – in general the workplace – and the context of that domain is crucial in the development of the interpretative process. As language is crucial to interpretation, the context on the domain will guide the language being developed. This language is crucial to the organisational learning process, as it is through this developed language that the individual will be able to communicate the new concept to the group level – integrating the knowledge.

Integrating can be defined as the collective action to evolve a shared understanding of the new concept. The primary inputs into integration are conversation and shared practice. As conversation plays such a key role in the integration process, the development of a correct language is crucial. It is essential that the individual's interpretative process has developed the language in a way that others within the group can relate to. The language will further be developed through the integration process, developing an "organisational language" relating to the new concepts.

Finally, institutionalisation can occur with group understanding and language relating to the new concepts. The group understanding can be used to develop and modify organisational practice to reflect the new knowledge. It is only with this development that the organisation can be considered to be knowledgeable in the new concept.

⁶ The concept that in order to assimilate knowledge one must link it to existing knowledge structures is often referred to as a "cognitive map" [6]

In summary, organisational learning is useful in considering technology adoption, as it helps in understanding how knowledge regarding an innovation is transferred throughout an organisation once the decision to adopt a technology has been taken. This complements and reinforces existing theories that only consider innovations transfer up to the point of adoption.

3.3 Summary

The above has reviewed research regarding the evaluation of software technologies, in particular the field of empirical software engineering, and also examined work in the use of theories of adoption for innovations. This provides a theoretical foundation for addressing the two issues identified at the start of the chapter, and each will be returned to throughout the thesis in developing a strategy for assessing component technologies and disseminating results.

As the focus of research is in two real world software projects, this final literature review chapter considers the case study approach, and the problems faced in using it. An understanding of both the strengths and weaknesses of such an approach provides the context for the method used in this research programme.

4. A Case Study Based Approach to the Assessment of Component Technologies

The choice of research method in this programme was constrained by the settings in which the work could be carried out. As the work was funded from two industrial projects, the focus of research work had to exist within the industrial setting. Thus, the research programme centred on work with component-oriented technologies within these two industrial software development projects. Under the circumstances, case study provides the most natural approach to the research.

However, this practical constraint can also be seen as advantageous in terms of literature from the viewpoint of both assessment (see section 3.1) and adoption (see section 3.2). The discussion regarding empirical software engineering and available techniques has highlighted problems in the transfer to industry of research results from laboratory experiments. Criticism has identified practitioner mistrust with such approaches, as they do not have their foundations in the “real world”. As the development of results in this research programme aimed to share the experiences and findings of the project in order to develop better practice in the development of software from components, it seems sensible to base the technology assessment within a practitioner oriented context. The scale afforded in these cases would be impossible to replicate within an artificial situation. Also, the programme cannot be considered a process improvement project, as aspects such as measurement and definition of process were beyond the control of the author. However, contributions from that area, in particular the Quality Improvement Paradigm [11], provided good

background when considering the approach to the development of results from the assessment of component technologies.

In considering the research programme from the adoption perspective, component technologies are still very new. If we refer back to Redwine & Riddle's [127] comment regarding the time taken for an innovation to reach maturity (i.e. fifteen to twenty years) we can consider component technologies to still be very immature. Again, such immaturity can benefit from a case study based approach. A lot of literature (for example, Yin [166] or Benbasat, Goldstien & Meand [16]) focuses upon research where the relationship between phenomenon and context are not clearly defined.

From this discussion we can conclude that while the choice of research method was restricted, the nature of the technology to be assessed means that a case study approach is an effective one. In order to discuss the nature of the case studies within the research method context, it is necessary to define them. A brief overview of each is therefore included here, greater detail is provided in chapters 5 and 6.

Case study 1: DOLMEN – A pan-european consortium of telecommunications service providers and software houses that developed an Integrated Service Environment (ISE) to incorporate fixed and mobile telecommunications technologies into a common telecommunications platform. The use of component technologies aimed to simplify the development process and provide a library of reusable telecommunications components for future work.

Case Study 2: Netscient – A network management independent software vendor (ISV) in their first year of business, wishing to incorporate component technologies into their development processes to improve productivity and promote a reuse culture within the organisation.

In each case the use of component technologies was considered based upon different forms of evidence and initial conclusions for each case were drawn. As the case studies were carried out sequentially some findings from case study 1 were fed into case study 2 in order to test them in a different context.

The cases provided a good opportunity to study component-oriented techniques within software development projects. However, case study findings are often difficult to generalise. Therefore, while the second case study provided some opportunity to test initial findings, further validation was sought through a practitioner survey, in which a questionnaire was constructed based upon case study findings and presented to other practitioners with component technology experience. This provided a good opportunity to test the theories developed from the case studies, and also to guide the development of results in a more practitioner focused way. The survey construction, delivery and findings are presented in chapter 7. Figure 4-1 provides an illustration of the research approach as a whole.

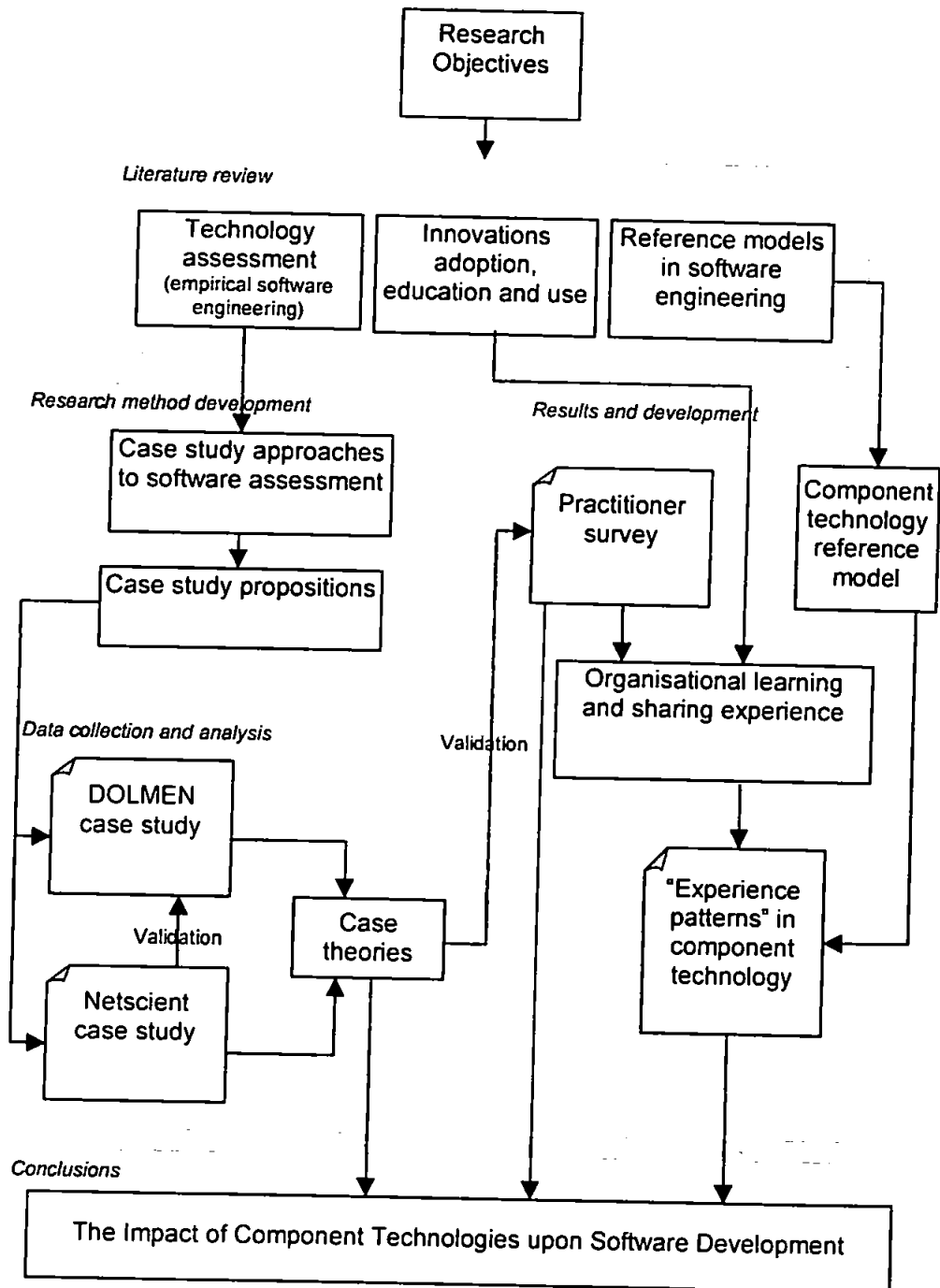


Figure 4-1 - Roadmap of Research

4.1 A Review of Case Study Research

Case study research is an approach that arose within the social sciences. The most commonly cited work in this area comes from Robert Yin [165].

A technical definition of the case study by Yin is stated as:

"An empirical inquiry that

- investigates a contemporary phenomenon within its real life context, especially when*
- the boundaries between phenomenon and context are not clearly evident." (pp. 15)*

And states that the case study inquiry

- copes with the technically distinctive situation in which there will be many more variables of interest than data points, and as one result
- relies on multiple sources of evidence, with data needing to converge in a triangulating fashion, and as another result
- benefits from the prior development of theoretical propositions to guide data collection and analysis.

The case study approach is an all-encompassing method – it defines both data collection and analysis strategies (see sections 4.1.3 and 4.1.4). It also, however, relies on clear aims and propositions prior to its execution in order to extract the correct data from the data set (i.e. the case itself). The following reviews the major elements of the case study research approach.

4.1.1 The Research Design

The research design, stated colloquially by Yin, is

"an action plan for getting from here to there, where here may be a set of initial questions, and there is some set of conclusions. " (pp 19)

It defines the boundaries of the investigation, so evidence can be focussed upon addressing research questions. Five components are defined:

- **A study's questions** – Yin suggests that case studies are most suited when the nature of questions related to "how" and "why".
- **A study's propositions** – the propositions of a study allow focus within the study questions
- **Its unit(s) of analysis** – the most crucial aspect of the case study method, as it relates to the problem of defining what the case is. In general, the unit of analysis is developed from the way the initial questions are defined
- **The logic linking the data to the propositions** – how the collected data (see section 4.1.3) is analysed to relate back to the research questions
- **The criteria for interpreting the findings** – how this analysis is developed into results and conclusions.

4.1.2 Types of Case Study Designs

Taken from [166], Figure 4-2 depicts the four basic types of case study design. The difference between single- and multiple-case designs should be clear, but the difference between holistic and embedded cases merits further clarification. A holistic design focuses upon a single unit of analysis, for example assessing the effect of a given procedure upon an organisation. The holistic view would only examine the impact upon the organisation as a whole. Embedded cases have

further subunits within the overall unit of analysis, enabling analysis of particular aspects of the unit. To use the above example, embedded units within the organisation could be individual departments.

	Single case	Multiple cases
Holistic (single unit of analysis)	Type 1	Type 2
Embedded (multiple units of analysis)	Type 3	Type 4

Figure 4-2 – Basic types of designs for case studies

4.1.3 Data Collection

Six types of evidence are defined as suitable in the case study approach:

1. **Documentation** – this can take many forms – letters, memos, other communication (for example, email), project reports, administrative documents, etc. and is useful to clarify information and motive, develop inferences, prompt further investigation, etc.
2. **Archival records** – similar in use and style to documentation, but is generally more formal and exact (for example, project deliverables).
3. **Interviews** – a way of focusing on exacts within the case, they can also help in further investigation of points that have arisen from other sources of evidence.

4. **Direct observation** – executed by case study workers “on site”, direct observation does not affect the case as it is being carried out but reflects upon practices, behaviours, etc.
5. **Participant observation** – a more active form of observation, when the observer assumes a role within the case and participates in the events being studied. Participation provides unparalleled opportunity to access “insider” information about the case.
6. **Physical artefacts** – a form of physical evidence that can be collected and studied away from the case site.

Data collection should follow 3 principles:

1. **Use multiple sources** - enabling a *convergence of lines of enquiry* or, *triangulation*, so that conclusions can be drawn from a number of different sources and are, therefore, more reliable.
2. **Create a case study database** – in order to organise and document the data collected. In a long case study with considerable amounts of evidence, it would be very easy to lose track of evidence being collected unless it is done so in an organised way.
3. **Maintain a chain of evidence** – in order to increase the reliability of the evidence, and so that external reviewer can follow inferences made by investigators.

4.1.4 Data analysis

General analytical strategies focus upon either following the case’s *theoretical propositions* – the statements at the beginning of the study upon which the research questions are developed – or through it structuring in a *case description* – a framework upon which the case can be organised. Within the analytical strategy, it is necessary to use specific analytical techniques. Yin defines

two modes of analytical technique – dominant modes that deal with internal and external validity, and lesser modes that generally have to be used in conjunction with a dominant analytical mode.

4.1.4.1 Dominant modes of analysis

Pattern matching – comparing empirical patterns (i.e. those drawn from the case) with predicted ones. By defining a predicted pattern based upon variables within the case before the case is carried out, some measurement can be made regarding the outcomes of the case based upon these predictions.

Explanation building – analysing the case study by building an explanation about the case, identifying causal effect that relate to elements of the case.

Time series analysis – examining changes in case study variables over time to demonstrate conditions one or several outcomes.

Program logic models – a combination of pattern matching and time series analysis to build a chain of events over time, therefore demonstrating the outcome of the case based upon the causal relationships between the events over time.

4.1.4.2 Lesser modes of analysis

Analysing embedded units – applying a technique pertinent to the embedded unit of analysis whose conclusions can then feed into the propositions for the whole case

Making repeated observations – carrying out a similar set of observations at varying times, to identify commonly occurring events, etc.

Making a case survey – in the event of several case studies a surveying of all studies to assess case outcomes based on standard measurements.

4.1.4.3 Assessing the Quality of the Research Design

Finally, consideration needs to be made regarding the effectiveness of the case study – essentially the quality of the study and its results. Yin defines validity measures similar to those defined by Basili et. al [12]. However, he also defines an additional measure of research design quality – reliability. Yin's definition [166] for all four measures are defined below:

Construct validity: establishing correct operational measures for the concepts being studied

Internal validity: establishing a causal relationship, whereby certain conditions are shown to lead to other conditions, as distinguished from spurious relationships.

External validity: establishing the extent to which a study's findings can be generalised

Reliability: demonstrating that the operations of a study – such as the data collection procedures – can be repeated, with the same results.

4.2 Relating the Case Study Approach with the Research Method

While the case study approach comes from a social science background, its use in the study of phenomena with the field of IT is growing, sometimes implicitly [53], but also explicitly [16, 107, 92]. An early paper on the subject by Benbasat et. al. [16] suggested that idiographic research (understanding a phenomena in context) rather than nomothetic methods (laboratory research) was preferable in the information systems field. They stated that case study approaches were particularly preferable where research is at an early stage or when practitioner experience within context would provide important findings. They provide three reasons why information systems research can benefit from case study research strategies:

1. The systems can be studied in their natural setting and theories can be generated from practice

2. The case study allows the research to ask “how” and “why” about the processes taking place
– leading to understanding of the nature and complexity of the study topic
3. A case study is appropriate where an area has had few previous studies carried out.

The study by Murphy, Walker and Baniassad [107], comparing the use of case study and experimental techniques to assess emerging software development technologies, considered the case study approach suitable when the broad effects of the impact of the technology were of primary interest and when identifying and addressing usability issues.

The closest in approach to the method in this research programme is work published by Kunda and Brooks [92]. In this research, the authors examine the socio-technical effects of using component-based development. The research is of particular interest as it demonstrates good case study practice in a similar area, but with different case study propositions, and with a different case study protocol. Also, in an area such as component-oriented development, where the majority of literature is industrial in nature, it also provides a compatible study to compare case studies from this research programme.

In relating the case study approach to this research programme, in terms of the definition of a case study from section 4.1, we have both a contemporary phenomenon within the field of software engineering – the emerging technology of component-oriented development - and a context for this phenomenon – the development process within which the technologies are used.

In considering the value of a case study approach to the field when considering the reasons for case study research put forward by Benbasat et. al. [16], all three reasons can be seen to be appropriate in the aims of this research programme.

4.2.1 Defining the Research Approach in terms of Case Study Research

The definition of the aims and objectives of the research as stated in section 1.2 clearly identify our core unit of analysis – the software development process. However, within that unit of analysis, we also wished to consider the effect of the technologies upon individual activities within the development process – these activities (analysis, design, etc.) become the embedded units of analysis within the research design.

While at first we may consider the research design to be that of type 4 from Figure 4-2, it is actually, two separate single case studies investigating the same phenomena. Each case has a similar aim, i.e. to investigate the effect of component technologies upon the underlying development process, and in each case similar data collection techniques were used. However, the difference in the context of each case meant that the studies were not directly comparable or differing in a controlled, predictable way. Therefore, they cannot be considered part of the same “study”. This difference in context can, however, be exploited - by using the two cases as individual case studies, we can define different propositions for each in order to develop our understanding of the implications of using component technology. Therefore, while each case study is exploratory in nature and addresses similar aims, the case propositions allow a focus of these overall aims in each case:

4.2.1.1 General Case Propositions

- Adopting and using component technologies in software development processes will affect process activities
- An awareness of the issues involved in the adoption and use of component technologies can ease their integration

4.2.1.2 DOLMEN Case Propositions

- Component technologies ease the development, integration and deployment of distributed systems
- Uncontrolled adoption and use of component technologies can have a negative affect upon a development project

4.2.1.3 Netsciënt Case Propositions

- A domain-oriented approach to component development provides a greater degree of reuse than a product oriented view.
- Similar issues with component-orientation occur when using different technologies from the same field (i.e. Microsoft based, rather than OMG based technologies)
- Issues in the DOLMEN case study can be avoided through greater knowledge of the technologies involved

4.2.2 Data collection techniques

Of the six types of evidence discussed in section 4.1.3, five were used to varying degrees within the case studies of this research programme. While each case study (see chapters 5 and 6) provide a detailed definition of sources of evidence, general types, in relation to those defined by Yin [166], are listed below:

Archival records – project deliverables, specifications, etc.

Documentation – internal reports, emails, memos, design documentation, etc.

Direct observation – used in assessing the effect of management aspects of the development process

Participant observation – from within participative roles in each case (see the discussion of roles within the cases below).

Interview – either in person, or through telephone or email conversation with key personnel in each case.

Of these types, the final three (direct observation, participant observation, and interview) were the main sources of evidences, supported through the other two. In order to differentiate between the different types of evidence collected, it is important to define the author's role within the execution of each case –each case study chapter provides a detailed description of the participative role. It should also be noted that in each of the studies, the author played two roles, one as an observer and one as a participant.

4.2.3 Data analysis techniques

The general strategy for data analysis was to base it against both the general and case propositions. These propositions provided a qualitative measure against which the case findings could be measured. The predictions from the propositions could either be demonstrated or rejected based upon the evidence collected. In addition to this pattern matching technique, embedded unit analysis also enabled a focus of attention upon the individual activities within the development process. For each embedded unit, case propositions could be tested against the evidence collected.

4.2.4 Case Study Reporting

Finally, the reporting of each case is structured based upon a definition of aspects of each case, followed by a review of software development identifying issues arising from the use of component technologies. Each case follows a similar structure, discussed at a study specific level in the relevant chapters, but presented in general terms below:

1. **A Definition of the Software Development Process** – The unit of analysis in each case, the development process is defined at a high level. It is important to understand that the use of the technology occurred within a development process, which could have an affect on the application of the component techniques. The definition is used as a point of reference in the assessment of the technology.
2. **A Definition of the Component Platform** – defining the choice of component technologies (and other software technologies) used in the case, again as a point of reference when considering case study outcomes. The model used to define the component platform in each case is defined in section 8.2.
3. **Case Study Analysis** – identifying issues arising from the development of software using component technologies. Consideration is made toward all development activities, and issues for analysis are identified.
4. **Case Study Results** – developing the analysis into a presentation of case study results, testing case study issues against case propositions in order to develop theories in the adoption and use of component technologies.

4.2.5 External Validity and Reliability in the Research Method

When considering the quality of the research method, we focus mainly upon external validity and reliability. Construct validity is demonstrated through the definition of propositions and strengthened through multiple sources of evidence. Internal validity relates to causal relationships, and is therefore not an issue in these cases. However, external validity and reliability are both crucial in assessing this research approach – it is expected that the case study results will be used by other projects when attempting to use component technologies. Therefore, it is important to address the issue of generalisation from the case findings. Just because events occurred within the case studies, can we assume they will happen in other cases?

It is, however, worth noting the difference in generalisation in the *statistical* and *analytical* sense. In a more traditional approach to gaining evidence relating to a phenomenon, generally through surveying techniques, the experimenters are attempting to state that if something occurs in a sample, it is a given that it will be proportionally reflected (generalised) in the wider universe. This is, of course, not possible from a single case study, where evidence will generally not be quantifiable. However, an analytical generalisation does enable the application of the findings to the development of a theory that can be tested through further case study or other analytical approaches. The concept of developing theories from case study research is dealt with explicitly by Eisenhardt [48]. Figure 4-3 is developed from that paper, and illustrated the process of developing theory from case study research.

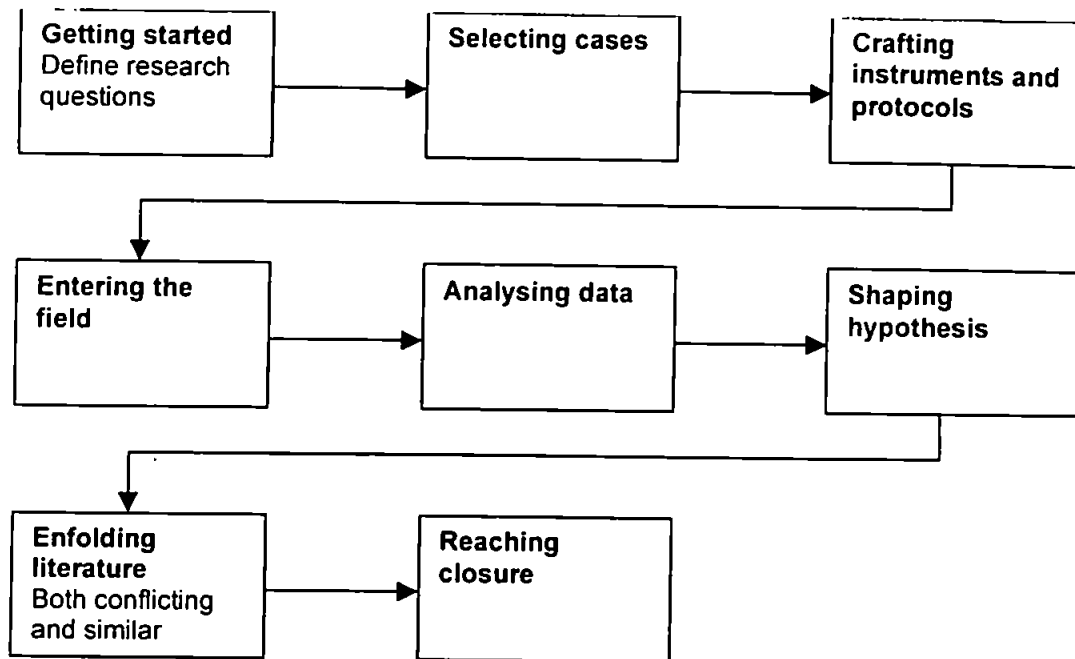


Figure 4-3 – The process of developing theory from case studies

Basil & Lanubile [8] define a theory as “a possible explanation of some phenomenon”. This issue is dealt with in two ways within this research programme. Both case studies develop theories in the affect of component-orientation upon software development. In each case events that occurred through the use of component technologies are explained based upon evidence from the case itself. Theories developed from the first case study are initially tested against the second case study. Indeed, one of the case propositions for the Netscient case study is:

- *Issues in the DOLMEN case study can be avoided through greater knowledge of the technologies involved.*

However, additional theories are also developed from the second case study. Therefore, the practitioner survey carried out following the case studies tests theories developed from both case studies, to focus upon generalisable findings, and to guide further explanation regarding theories.

The issue of reliability is also important in the development of results from the case studies. Could a future investigator draw similar conclusions from the evidence presented in the case studies and are the conclusions free from error or bias? The issue of bias is addressed through backing up opinion presented with further evidence, either from observation or from evidence obtained from other case participants. The database of case evidence and use of multiple sources of evidence also aids in the reliability of the cases.

4.3 Summary

This research programme aimed to investigate the effect of component technologies upon the software development process and, through this investigation, implement techniques to aid in the good practice of component-oriented software development. This chapter has detailed the research method used in collecting, analysis and developing data. The primary research approach is through the examination of the adoption and use of the techniques in two cases – the approach can be associated with the case study research method, as defined by Robert Yin [166]. Additionally, a further survey of the field strengthens the validity of the case findings. The approach applied to specific case instances is discussed in chapters 5 and 6, and the survey approach and findings are discussed in chapter 7.

The next three chapters describe the data collection, analysis and initial results presentation from the research programme. This chapter, the first of the three, details the first case study, using a CORBA approach within the telecommunications domain. Drawing from previous chapters discussing technology assessment and case study approaches, the study is guided by case propositions based upon research aims. Discussion of the research method in this case is included to guide the reader in understanding the results that come from the study.

5. Software Components in the Telecommunications Domain

5.1 An Overview of the DOLMEN Project

The European Commission funded project DOLMEN (Service Machine Development for an Open Long-term Mobile and Fixed Network Environment) was a telecommunications architecture project in the ACTS (Advanced Communications, Technologies and Services) programme. The project, which ran from 1995 to 1999, was based in Integrated Services Engineering, following a growing trend in telecommunications (for example, the international Telecommunications Information Networking Architecture (TINA) standard [156]) to move away from the traditional approach of hardware controlled switching systems to a more flexible software controllable telecommunication network, across heterogeneous technologies and multi-provider environments. The aim was to develop work from Intelligent Networks (IN) [1] and Telecommunications Management Networks (TMN) [126] based around the idea of providing multiple services over the same network. This is achieved through the isolation of the management network from the actual network hardware using a software controllable infrastructure for the control and management of services on top of the network. This provides the service developer with a standard platform upon which to create services, without having to worry about how to control the underlying hardware. The hardware control is taken care of by the software platform.

This in turn places a requirement on the architecture for the Distributed Processing Environment (DPE), which provides a platform upon which to implement the Service Architecture and enables distribution over many computers (known as Service Nodes). Figure 5-1 illustrates this concept.

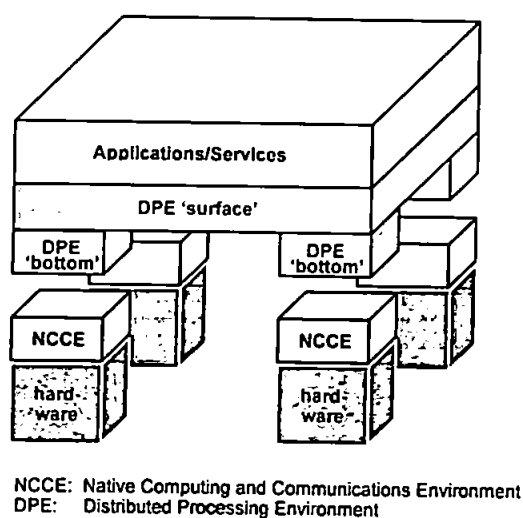


Figure 5-1 - An Illustration of an Integrated Services Environment

This is taken from a DOLMEN deliverable [30] and, therefore, the terminology is very telecommunications centric. Each “leg” of the diagram represents a computer, or service node, within the service environment, responsible for the control of a given piece of hardware. The NCCE (Native Computing and Communications Environment) can be viewed as the Service Node’s operating system / network operating system, and the DPE ‘bottom’ is the particular installed distributed processing software (for example, an implementation of CORBA or DCOM), which provides location, access and operating system transparency. Then, a client object makes a call without knowing where the server is located, how it is accessed or on what platform it executes – that is all resolved by the DPE [78]. Therefore, we can view the DPE as a whole - the DPE ‘surface’. Finally, it is the DPE surface that enables applications to write to any component within the ISE environment (the upper layers of Figure 5-1) without having to worry about the

location of that component. In reality this is resolved by the distributed processing software on the node, plus distributed processing middleware.

The aims of DOLMEN were defined by the project [158] as

... to develop, validate and promote a Service Architecture that encompasses the needs for services providing mobility across mixed mobile and fixed environments. This architecture has been called Open Service Architecture for design and provision of communications services and applications over an integrated fixed and Mobile communications environment - OSAM. Its foundations are to be found in RACE OSA Architecture (a legacy framework) and TINA. The focus has been on extending TINA with architectural support for mobility. In particular this has meant:

- *To extend TINA to explicitly encompass personal and terminal mobility.*
- *To develop a set of OSAM-conformant Service Components.*
- *To demonstrate OSAM in the DOLMEN Final Trial, by using: (a) an existing mobile technology (GSM data service) and forthcoming mobile technology (UMTS)⁷, and (b) two applications (Audio Call Service and Hypermedia Information Browsing) to exercise the OSAM Service Machine.*
- *To promote OSAM within and outside the ACTS Programme, in particular towards global fora addressed by TINA-C, in co-operation with their Core Team.*

The development and demonstration was realised through the specification of layers of software modules, or components, to achieve service requirements. These requirements ranged from application functionality, through session management, service management and communication management to the lowest level of hardware control. Each component had a role in the environment, or (to use a DOLMEN term) Service Machine, and was able to communicate with other components in the architecture to carry out its function. For example, if an audio conferencing application required a high capacity communication link it would state its intentions

⁷ DOLMEN used radio-access emulated by wireless LAN as UMTS radio access systems were not available before the Final Trial.

to session control components. The session controllers, in turn, would know how to invoke this using communication control components. These would then break the overall request into parts of the connection, and ask those components that controlled that particular piece of the network to make the necessary connections. Finally, it would fall to hardware control components to set up the hardware in the appropriate way. The outcome would be communicated back up through the architecture to the application, enabling it to use the communications link.

5.1.1 DOLMEN Organisation Structure

A pan-European consortium undertook the DOLMEN project, each bringing specific skills. The twelve partners included:

- Telco operators;
- Manufacturers;
- Value added service providers;
- R & D institutes;
- Universities.

A more detailed overview of the DOLMEN project structure can be found in appendix A.

Partner personnel were divided into project workgroups and within the workgroups, workpackages. Each workpackage was responsible for a given aspect of the project development.

Figure 5-2 illustrates the workgroup and workpackage structure:

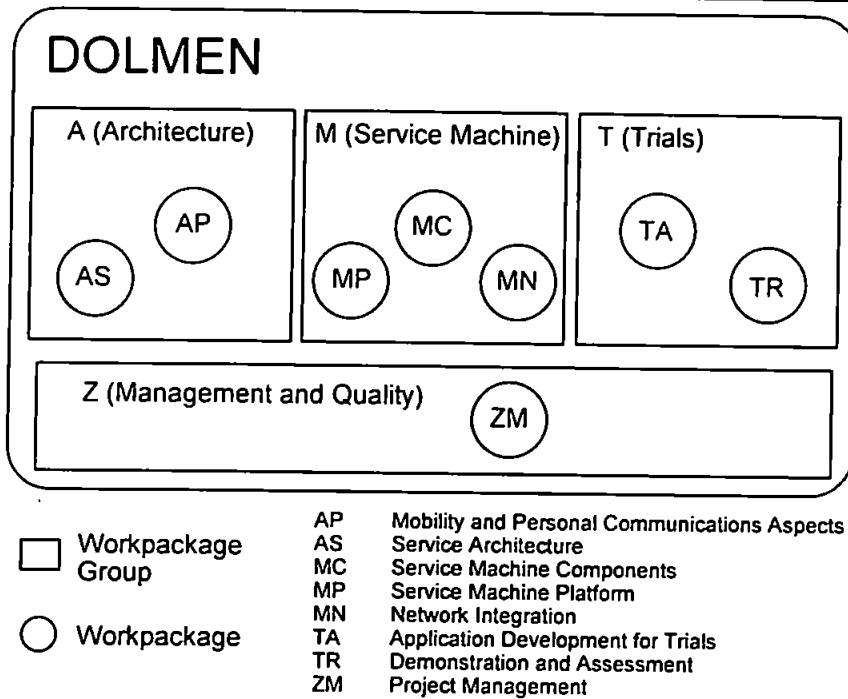


Figure 5-2 - DOLMEN Workpackage Structure

The following briefly defines each element:

Workgroup A: Responsible for the definition of the DOLMEN architecture, broken into:

Workpackage AP: which was tasked with the mobility and personal communication aspects of the architecture (e.g. integration of mobile technologies) [160,161]

Workpackage AS: which had the overall responsibility for the definition of the DOLMEN service architecture.

Workgroup M: Responsible for the development of the DOLMEN Service Machine, which demonstrated the architecture, broken into:

Workpackage MC: which was responsible for the specification and development of the service machine components

Workpackage MP: which was responsible for the specification and development of the Service Machine platform – the customised CORBA platform that integrated two ORB

products with some custom functionality to enable faster pan-mobile network object interaction.

Workpackage MN: which was responsible for network integration – the task of integrating the DOLMEN service components to specific network technologies through the development of *resource adapters*.

Workgroup T: Responsible for trial execution, broken into:

Workpackage TA: which developed the trial applications to demonstrate the Service Machine functionality.

Workpackage TR: which dealt with the development of the trial hardware and software configuration and the integration of all developed DOLMEN software (from MC, MN, MP and TA) into the trial environment.

Workgroup Z: Responsible for project management, consisting of a single workpackage (ZM).

As DOLMEN aimed to demonstrate aspects of a Europe wide telecommunications architecture, it was necessary to have an international aspect for the trial. Therefore, the trial configuration was split between two *national host sites*, one in the UK (provided by Orange Personal Communication Services) and one in Finland (provided by Telecom Finland). Each national host site provided a mobile and broadband network technology on which to demonstrate the DOLMEN architecture (Wireless LAN and ATM in Finland and GSM and ATM in the UK). Figure 5-3 provides a simplistic illustration of the trial organisational viewpoint.

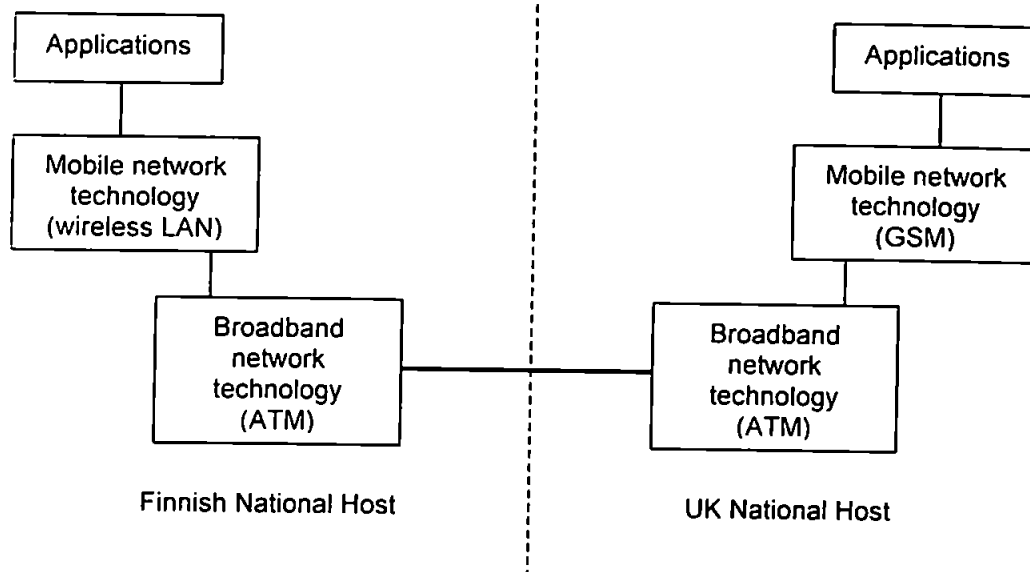


Figure 5-3 –DOLMEN Trial Set-up

5.1.2 The Use of Component Technologies in DOLMEN

In general, the model for an Integrated Services Environment is highly distributed. The hardware in a broadband telecommunications network can be distributed across a wide geographical area. In order that software can interface with the hardware to carry out user requests for network connections, it is necessary for the software at the lowest level of the environment to reside in the same geographical location as the hardware. A distributed software standard is therefore necessary to communicate user requests from the application level to the hardware control components. A feasibility study of distributed software platforms was conducted by workpackage MP [129] at the start of the project to assess the suitability of such for the DOLMEN project. The study assessed three different standards (DCE [112], ANSA [66] and CORBA). At the time of the study (1996), DCOM was not considered an option because it was still an immature product and it was only available on Windows platforms. As the overwhelming majority of telecommunications management systems use UNIX, it was not feasible to use a Microsoft

platform. The study concluded that CORBA was the most suited to DOLMEN's needs for the following reasons:

- it supported the majority of service machine criteria defined by the project (distribution, object-based, etc.);
- it was being taken up by the industrial community;
- a large number of CORBA compliant products were available;
- mobility requirements would be met easily using interoperability protocols [114];
- developing CORBA compliant components would enable integration with other ACTS projects in the same area (ReTINA [128], VITAL[162]);
- some CORBA implementations were available for Linux – the chosen operating system for some aspects of the DOLMEN project. As all products were CORBA 2 compliant, interoperability between different operating system implementations would be straightforward.

Once the decision to use CORBA was taken, the majority of software development toward the DOLMEN trials centred on the design and implementation of CORBA components and clients. The MP workpackage also developed some low-level enhancements to CORBA implementations to improve interoperability between ORBs.

As part of the study, work was also carried out to assess the potential of design techniques for both static and dynamic modelling of the service machine, to aid in the efficient development and integration of software components necessary for the DOLMEN service machine. The study concluded that the use of Structured Definition Language (SDL) [81] and Message Sequencing Charts (MSCs) [80], both standards from the International Telecommunication Union (ITU),

would aid the dynamic modelling of the system. Another study, by workpackage MC [14], concluded that the use of OMG-IDL (hopefully a 'given' in a CORBA project) and Object Modelling Technique (OMT) [135] would be useful in the specification of components.

5.2 The DOLMEN Case Study

While the project goals can be used as an indication of the relative success in the use of component based software development, it is important to differentiate between project goals and case study goals. The case study goals aim to assess the effect that the choice of component orientation as the chosen development approach had in the project. Therefore, while a lot of the aims of the project do not complement this assessment, there are a number of aspects that make it invaluable:

- it was a large, distributed software project using CORBA as the core software interaction standard;
- it was intending to develop a component suite which could be reused in other projects;
- it was attempting to develop a component-oriented system within a specific vertical domain (telecommunications / ISE).

5.2.1 Case Study Definition

Chapter 4 has discussed the general approach to the case study and the research methods used. This section examines issues specific to the DOLMEN case study. Firstly, it reviews case study propositions before elaborating upon the analysis approach, discussing strategy and types of evidence used. Finally, it defines the structure for the case study report, which makes up the remainder of this chapter.

5.2.2 Case Study Propositions

The propositions for the DOLMEN case study comprise both general case propositions and also DOLMEN specific propositions, defined in section 4.2.1, and repeated below:

- Adopting and using component technologies in software development processes will affect process activities
- An awareness of the issues involved in the adoption and use of component technologies can ease their integration
- Component technologies ease the development, integration and deployment of distributed systems
- Uncontrolled adoption and use of component technologies can have a negative affect upon a development project

5.2.3 Case Study Role

Partial funding for this research programme came from the DOLMEN consortium. This funding was on the understanding that the researcher had a role as developed within the project. This development role provided the opportunity for participant observation within the project. The role centred on two development tasks, initial with workpackage MN (see Figure 5-2) and then in workpackage TA. In each case the research had a peer level relationship with other developers within the workpackage, and reported to the workpackage-leader. The workpackage-leader assigned tasks to different developers and was in control of the direction of work within the workpackage. The researcher's two main tasks in the project were analyse work relating the DOLMEN architecture to mobile network technologies, and developing an audio conferencing application that used the architecture.

The developer role was valuable in collecting evidence relating to the adoption and use of these techniques in the various activities within the development process. As well as direct experience of the technologies, by being involved in development teams, the author gained access to other development personnel in order to obtain informal evidence (informal emails, ad-hoc discussion, phone conversations, etc.) relating to their experiences. As these experiences sometimes went against the official opinion (and therefore, the documented one) offered by project management relating to the success or otherwise of these techniques, this access was invaluable. Evidence regarding project management issues was collected through direct observation, archival records, documentation and interview.

5.2.4 Analysis approach

The general analytical approach for the case study is to drive case study findings from case propositions. In terms of analytical technique, the case study focuses upon “explanation building” – trying to determine the reasons for outcomes. However, as the case study was exploratory, rather than explanatory in nature (see Yin [166]), the aim is to use data collected to develop theories for further examination, not to develop complete conclusions to the effect of component technologies upon the development process.

Type of evidence are defined below:

- **Participant observation** – in a role of software analyst/developer within the project, participant observation provided an invaluable insight into the effects of using component technologies within the DOLMEN development process. Evidence from participant observation was generally written up as annotations to documentation (in the case of project meetings, reaction to internal papers, etc.) or in a simple field notes format in the event of development experiences.

- **Direct observation** – evidence was collected in a similar way as participant observation, which generally came from project management issues or development with which participation was not possible. In this case, personnel that were involved in an incident were generally approach either in person or via email to clarify events.
- **Interview with project personnel** – in order to clarify issues or to get some in depth information on a particular aspect of project development (for example, the interview with the project technical leader). Two types of interview were used:
 - **In person** – face-to-face interview allowed for a semi-structured discussion of a particular aspect of the project
 - **Via email** – to pursue matters when it was not possible to talk with the project worker in person.
- **Documentation** – provided archival records of incidents, ideas, project decision and milestones. Numerous types of documentation were available:
 - **Project deliverables** – formal documents that represented project milestones. These were of limited use, but did provide documentary evidence of definitive statements and policy within the project.
 - **Internal working papers** – the project generated a large amount of internal papers that were used to communicate ideas among project members. These tended to be less formal than project deliverables and focused on a specific detail within the project (for example, a choice of development tool, a specification of a given component, etc.).
 - **Meeting notes** – meetings were generally minuted, but personal notes were also kept. This provided opportunity to review the decision making process within the project and also served as a recall aid for participant observation

- **Project email** – such as announcements, requests, etc. These were also used as recall aids for participant and direct observation

Examples of evidence used in the analysis of this case study are included in appendix B.

5.2.5 Case Study Review

1. **The DOLMEN Software Development Process** – reviewing the DOLMEN software development process, a point of reference when assessing the effect component technologies had upon software development in the project.
2. **The DOLMEN Component Platform** – defining the choice of component technologies) used in the DOLMEN project as their platform for software development. This serves as a term of reference for considering case study outcomes.
3. **Case Study Analysis** – identifying issues arising from both the development and trial aspects of the project and analysing the issues identified. Consideration is made to possible causes for each issue, based upon case study evidence.
4. **Case Study Results** – developing the issues identified from the development and trial review for consideration against case study propositions.

5.3 The DOLMEN Software Development Process

Figure 5-4 illustrates the model for the DOLMEN software development process.

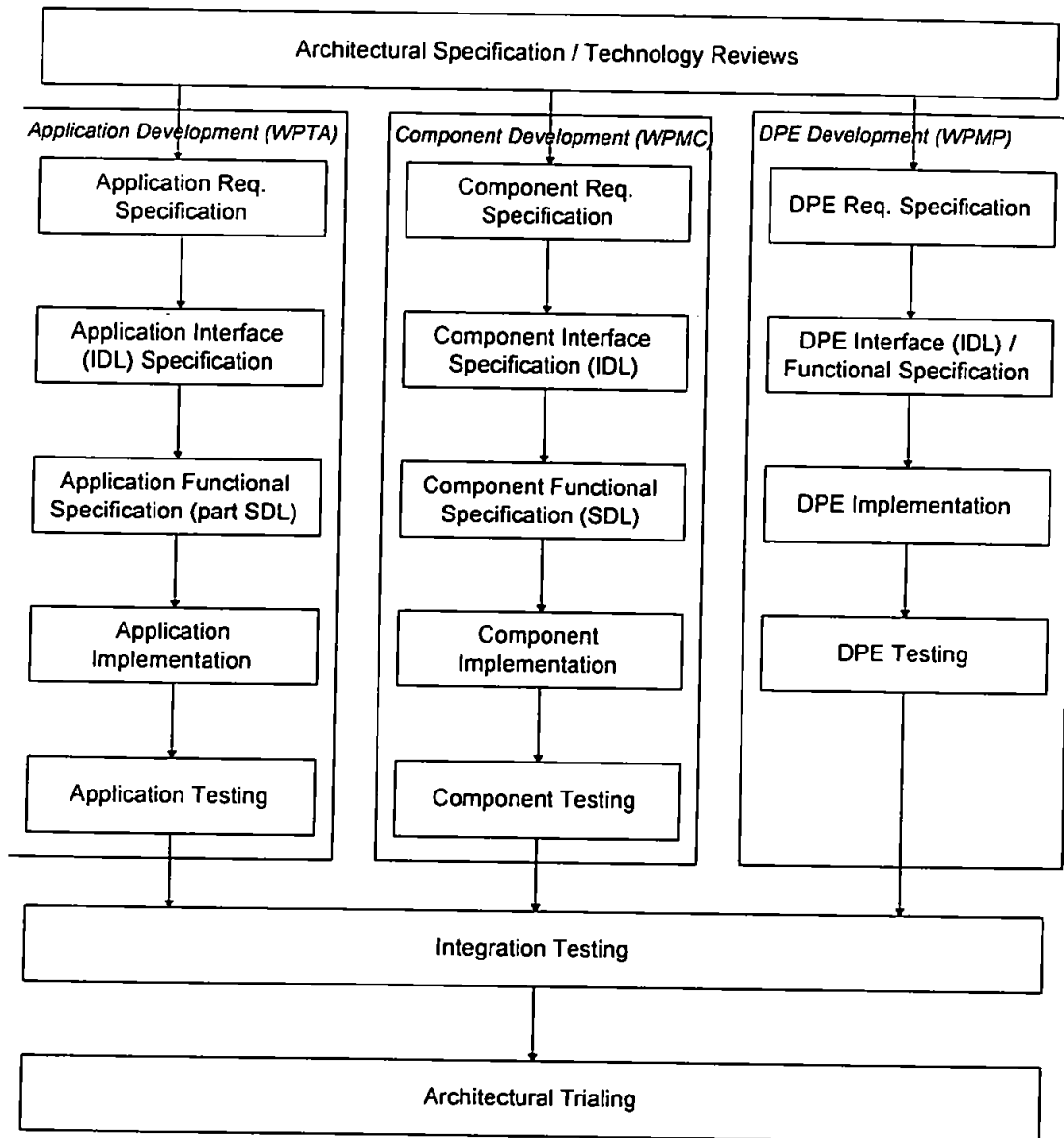


Figure 5-4 - The DOLMEN Software Development Process

Each aspect of the process is described below.

Architectural Specification: As the primary goal of the DOLMEN project was the definition and demonstration of an integrated services environment, the specification of the environment

was considered appropriate before the actual software development commenced. While architectural specification was ongoing, parallel activities assessed the feasibility of software techniques and technologies, as described above.

Core Development Areas: Software development fell into three main areas:

Application development: In order to assess the service machine, it was necessary to develop applications, or services, which would make use of the environment. Two applications were chosen: hypermedia information browsing (i.e. WWW browsing using an ISE rather than the TCP/IP standard) and two-way audio communication. Each exercised different aspects of the service machine. Information browsing used service-specific session management and some stream communication (for the downloading of data types requiring high capacity) while the audio application explored the more real-time aspects of the architecture (stream handover, real time data communication, etc.).

Component development: In DOLMEN, the entities that interact to achieve service machine requirements are termed components [158], but these are not software components as generally understood (while each can be considered similar in that they constitute an element of a system, DOLMEN components are defined in an architecturally specific way). The mapping from a DOLMEN component to software objects using CORBA was generally one to many, i.e., a co-operating group of CORBA components constitute a DOLMEN component.

DPE Development: The decision to use CORBA and the question of interoperability between CORBA implementations has already been discussed. Essentially, this issue

concerned mobile and fixed aspects of the service machine. On the mobile side the platform was Linux and Chorus CoolORB, while on the fixed side Solaris and Iona Orbix were used. Therefore, it became necessary to use an inter-ORB interoperability protocol [109] to communicate between CORBA implementations. The commonly used Internet Inter-ORB Protocol (IIOP) was considered too capacity intensive to be viable on a mobile communications link and, therefore, it was decided to develop a lightweight protocol (Lightweight Inter-ORB Protocol (LW-IOP) [129]) and CORBA services (for example, a federated naming service) which were usable by both ORB implementations. Thus, the DPE involved enhancement of CORBA products on both the mobile and fixed side to meet service machine requirements.

Core Development Phases: Within each development area, several development phases were recognised, along conventional lines:

Requirement specification: an assessment of the required functionality, developing architectural requirements toward a realisable technical solution. The majority of requirement specification involved the internal publication of requirements documents for peer group review, and developing the requirements documents into specification deliverables ([123,31,32,160,161,59])

Interface specification: developing the requirements specification into a static model to specify public functions and properties using OMT (in some cases) and OMG-IDL to formally specify the interfaces for each DOLMEN component. The publication of interfaces was considered the core definition of functionality between components in the

system – component clients (in most cases these were other DOLMEN components) used the defined interfaces to compile client calls into the developing code.

Functional specification: The dynamic modelling of the system was intended to make the actual implementation stage as straightforward as possible. By using modelling techniques the intention was to ensure all interactions between components were identified and specified before implementation. The modelling of behaviour between components was carried out by building functional models in SDL, which could be run through using design tools to identify problems with current models, until such time that all inter-component communication could be executed as a complete model. The SDL tools were also used to produce MSCs for various scenarios between components (for example, setting up an access session, requesting a stream connection between two parties, etc.). The publication of MSCs and some SDL models [31,32] provided a specification of functional behaviour for the DOLMEN components. In the case of the DPE, SDL was not used. However, MSCs were generated by hand to identify interactions between components.

Implementation: Implementation transferred interface and functional specifications into CORBA components. For the majority of implementation C++ was used, but Java was employed in some parts of the information browsing application.

Testing: Following implementation, local testing was intended to ensure component implementations were fully functional and bug free before integration testing.

Integration testing: Incorporated the developed applications, components and DPE into the DOLMEN service environment. As trialing was to take place between national host sites in the UK and Finland, integration was carried out at these locations.

Architectural trialing: Following integration, trialing carried out assessment of the functionality and performance of the DOLMEN architecture based on scenarios developed for the trial (for example, local interactions, international interactions, mobile aspects, etc.) [69]. The final conclusions of the trialing could then be used to both validate the architecture and influence future work in the area.

5.4 *The DOLMEN Component Platform*

The combination of component standards and services used in DOLMEN is shown in Figure 5-5. The salient features are as follows:

The Component Standard: The chosen component standard for the DOLMEN project was CORBA. The DOLMEN ORB as a whole comprised both Chorus CoolORB and Iona Orbix implementations; CoolORB in mobile domains and Orbix in fixed domains. Interoperability between CoolORB and Orbix was resolved using the Lightweight Inter-ORB protocol implemented in the bridging service, described below.

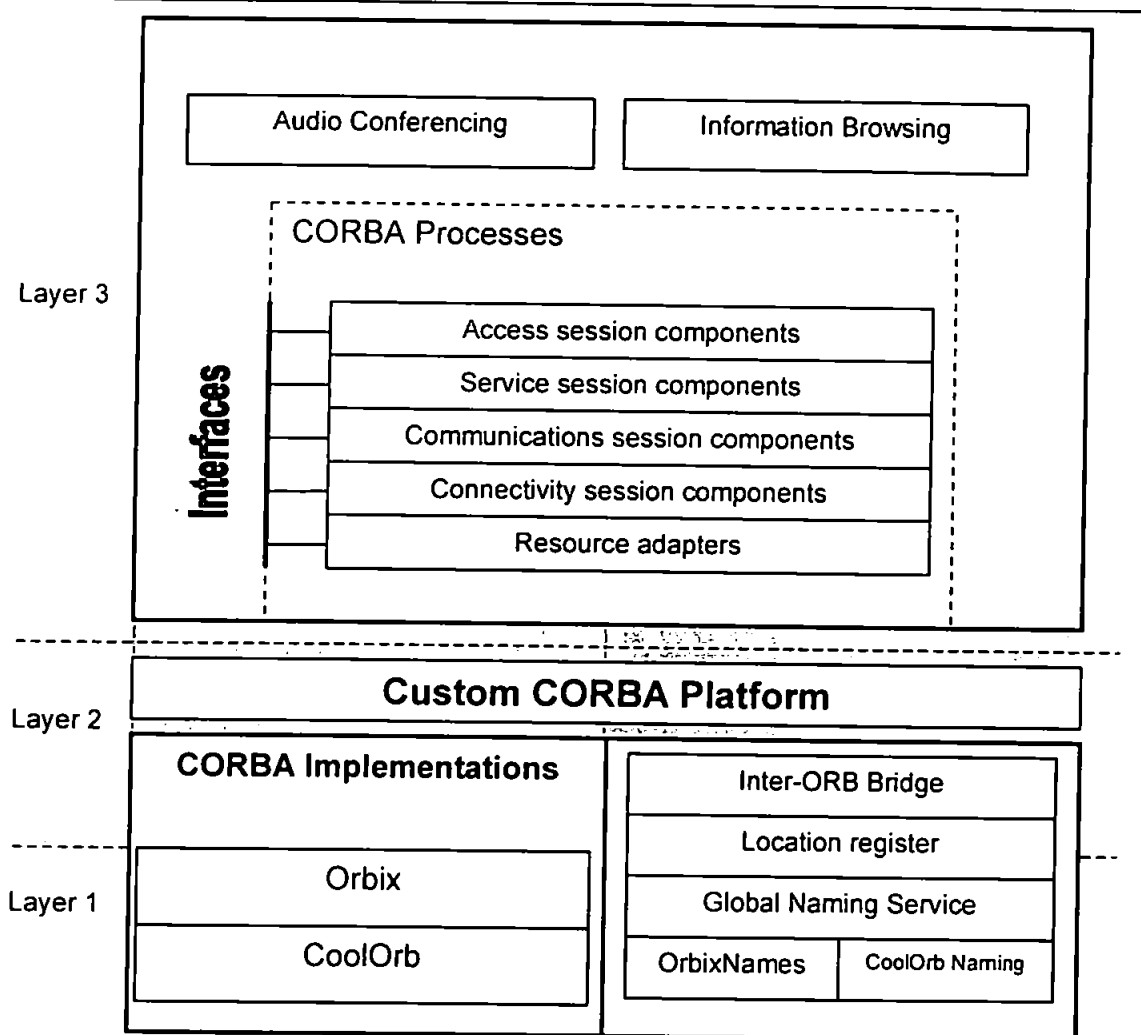


Figure 5-5 - DOLMEN Component Platform

Component services: The services supported within the DOLMEN component platform are provided to primarily enable the transparent integration of mobile and fixed ORBs. Three services are defined:

Naming service: The Global Naming Service (GNS) is the only one that can be considered a standard CORBA service. It enables a component within the platform to obtain the name of any object, whether it resides in a fixed or mobile domain. This is

achieved by integrating CoolORB and Orbix clients, via a GNSClient API, to the OrbixNames naming service.

Location register: This enables the ORB to keep track of the location of components in the platform. This is necessary because mobility adds the potential for components to roam different IP addresses, the mechanism that is generally used by a CORBA ORB to locate objects.

Inter-ORB Bridge: The bridging of the different ORBs across low capacity mobile links has already been discussed. It is the role of the bridge to implement the functionality that takes ORB requests, converts them into lightweight form (LW-IOP), transmits them, and unpacks and translates the call on the other side of the bridge.

Components: The DOLMEN components themselves provide the functionality of the DOLMEN service architecture, performing various roles to achieve the setting up and use of sessions within the service architecture. They are briefly described here and presented in detail in [31,32]

Resource adapters: take the interconnection requests and interface with network hardware to transform these requests into actual connections.

Connectivity session components: resolve the specific interconnections that are required to achieve the whole connection. The connectivity session resolves the sub network connections that are required to achieve the end to end connection requested by the communication session.

Communication session components: take service session requests and interpret the requests into a form that can be passed onto connectivity session components to achieve the requested connection. The communication session is responsible for end to end connectivity in achieving the required service.

Service session components: provide the service specific functionality within the architecture (i.e. information browsing or audio conferencing functionality). The service session establishes a user requirement for a specific facility offered by the DOLMEN architecture.

Access session components: Enable a user login to the service architecture, selection of service sessions, and management of roaming users [160,161]. The access session establishes a connection between a caller (user) and the DOLMEN Service Architecture.

Applications: Finally, the component clients for the DOLMEN component platform are end applications that exploit the service environment.

5.5 Case Study Analysis

On a project that introduced novel concepts in several areas, it was probably inevitable that some problems would arise when carrying out the software development. As it turned out, the entire process, from requirements definition onwards, was beset with problems which, when compounded, led to the project constantly battling schedules and struggling to meet defined goals.

5.5.1 Development Review

A common experience with software projects is that problems encountered early on in the development process can impact greatly on later phases. This was certainly the case in the DOLMEN project. The following reviews problems at each phase of the development process, from requirements definition through to implementation and testing.

5.5.1.1 Interface Definition Issues

Interface definition provided the first milestone in the specification of the system. The publication of interfaces was intended to enable client developers to compile calls to a server component without having possession of the component itself. As the component standard hides implementation details from users, having the interface should be enough to ensure that a client component will function properly with a server when they are integrated. Essentially the interface definition defines a contract between client developer (the service user) and server developer (the service provider). Therefore, in order that an interface definition be used effectively by client developers, one of the following must hold: either, the definition is frozen at publication, or, in the event of a change being required, it is properly documented and communicated to all development personnel.

In the case of DOLMEN, problems with interface definition emerged in a number of ways:

1. Immature interfaces were published as full definitions.
2. Dependent interface definitions (i.e. those included in other components, such structured types, enumerated types, etc.) were published and then modified and released as new versions.
3. Interfaces were revised and republished without communicating changes to client developers.
4. Various client developers used different versions of the same interface.

This had two obvious and serious consequences: client developer productivity was adversely affected, and the resulting software contained incompatibilities. This second failure was compounded by the rigidity of the development model (i.e. linear with no scheduling for iteration in the development process) which made it inevitable that problems would not become apparent until the implementation phase of the project.

5.5.1.2 Dynamic Modelling Issues

The intention of dynamic modelling of the system was to identify and test all inter-component interactions before implementation. The model developed in SDL certainly provided an effective demonstration of the DOLMEN component interactions from which MSCs could be demonstrated for all of the trial scenarios. However, problems in the use of such a technique became evident when attempting to map from specification to implementation.

SDL had its origins in the specification of embedded hardware systems, where the components can be modelled before being manufactured. However, in DOLMEN the aim was to develop a software system using component-oriented techniques to meet an architectural specification. The problem arose because DOLMEN components did not always map tidily onto a CORBA object. In most cases, a number of CORBA objects made up a single DOLMEN component. Therefore, while the SDL model aided greatly in observing the workings of the DOLMEN architecture and the interactions between DOLMEN components, it was not directly relevant to the construction of the system from implemented CORBA objects. Automatically generated MSCs (such as Figure 5-6) had to be greatly modified, or discarded altogether and produced by hand to enable the component developers to identify interactions between developed objects. The figure is included solely to demonstrate the nature of an MSC; the technical detail presented within it is unimportant. However, to briefly explain the structure, the chart demonstrates an interaction

between four interfaces (not components) within the service architecture. Arrows between interfaces represent function calls upon the interfaces and returns from them.

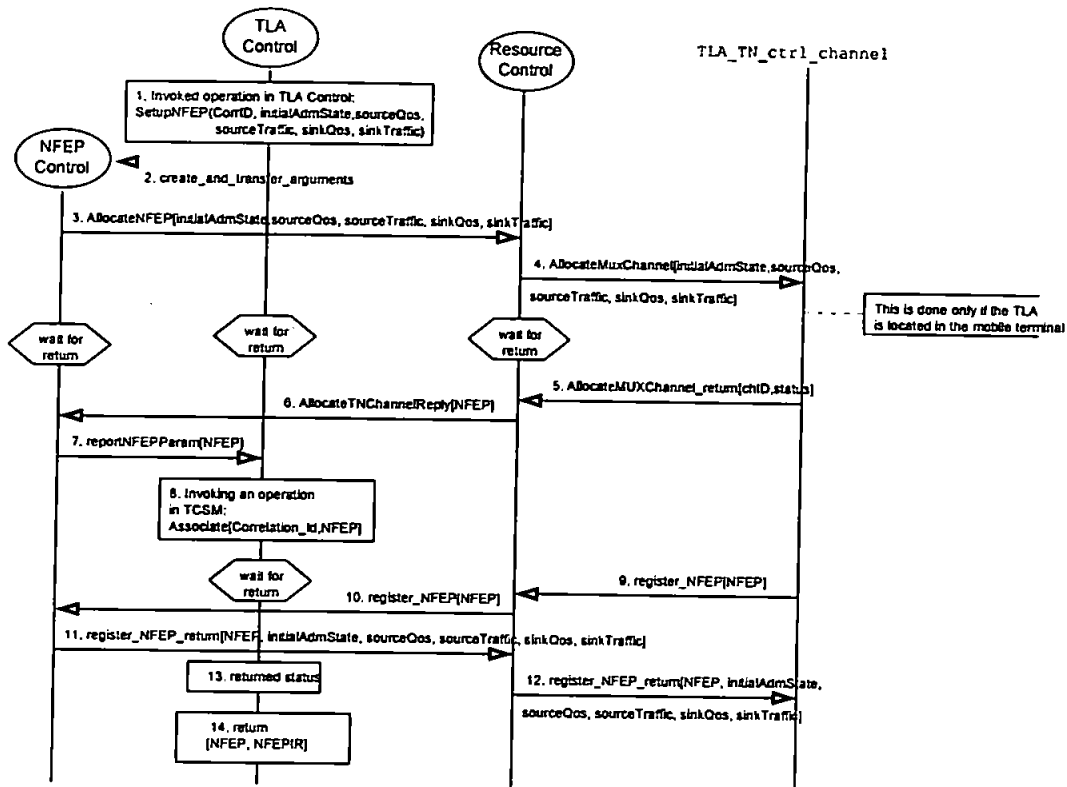


Figure 5-6 – A sample MSC showing component interfaces and interactions between them (taken from [59])

Once implementation-oriented MSCs were generated they were of great use to developers. However, as with interface definition, their utility was restricted by poor version control and poor communication regarding change.

5.5.1.3 Implementation Issues

Continually changing interface definitions and MSCs had an equally disruptive affect upon implementation. Private communication between developers of immediately dependent components (e.g. application and access session components, application and service session

components) enabled the resolution of inter-component communication at the sub-system level. Once interface definition and MSCs had been agreed between the two parties, developers could work in virtual isolation knowing that the dependent components were working to the same specifications. However, communication of changes outside the immediate group was typically in the form of technical reports, with some email announcement as to the availability of the documentation. With no central point of communication, it is hardly surprising that during integration testing change information did not always reach beyond the immediate group, as became more and more apparent as testing progressed.

The majority of problems with implementation were a consequence of earlier development phases, there were also two issues intrinsic to the implementation phase. These were problems with multiple interfaces, and problems with CORBA implementations.

The concept of multiple interfaces implemented by the same component class is found in both the software (e.g. the COM standard, Java, etc.) and telecommunications (e.g. TINA) domains. In DOLMEN, there was therefore an assumption that the chosen component implementation would support multiple interface definition. However, they are not in fact part of the CORBA 2 standard, and are accordingly not included in all CORBA products. As implementation approached it became apparent that while the CoolORB implementation provided the functionality to bind multiple interfaces to a single CORBA object, this was not true of Orbix. The situation was resolved through a project partner providing a mechanism that enabled such functionality in Orbix. However, it was another unforeseen issue that contributed to development problems.

It was also assumed that any ORB interoperability problems would be between implementation (i.e CoolOrb and Orbix). These were resolved as planned, by the development of a lightweight DPE that dealt with all inter-ORB communication. However, problems also arose in other areas, in particular between different language implementations of the same ORB (e.g. Orbix C++ and Orbix Java). This did not become apparent until the integration phase.

5.5.1.4 Testing Issues

Local testing was the final free-standing phase of each strand of development. It was included, as usual, to ensure that integration testing would be as straightforward as possible. However, many cases an isolated component offered little functionality – it was the collaboration with other components that provided the processing for a particular event (for example, a user login). In such cases local testing phase was of limited value. In reality, many developers:

- implemented their own dummy server objects and may or may not have used the same interface definitions as the actual implementation; and/or
- claimed a component had been fully tested when it would have been better to have said that it was tested as well as could be expected without dependent objects being available.

5.5.2 Trial Review

In this section we review a number of issues arising during integration testing and trialing. It was within this period that the majority of implementation problems came to light. Therefore, the process was by no means as straightforward as anticipated by the management team.

5.5.2.1 Integration Issues

The integration phase was where major problems with version control of both interface definition and functional specification became apparent. The resulting incompatibilities inevitably resulted

in changes in implementation, which impacted greatly on development schedules. The management team had allocated what seemed a reasonable amount of time (four months) for integration, and scheduled a number of integration workshops within that framework. However, the majority of workshop time was spent discovering problems that effectively halted any further integration testing until specifications could be agreed and re-implementation carried out. This it soon became apparent that integration would take far longer than estimated. As it turned out, it was not until the proposed final workshop before trialing that a full functional and interface specification review was carried out to produce definitive versions of the component interfaces, and complete MSCs for all trial scenarios.

Another problem that greatly hampered integration was delay in the delivery of components upon which others were dependent. For example, in one DOLMEN trial application, two elements of core functionality were required to establish an access session (i.e. logging on to the service machine) and then a service session (i.e. requesting service machine functionality for either an audio communication session or an information browsing session). Application developers were completely dependent on the necessary components being available in order to establish an access session and test service session functionality. Late delivery of such components resulted in a lot of wasted time in integration workshops. While, in hindsight, it seems obvious to identify such dependencies and develop schedules based on them, it was another problem that was not anticipated early on in the project.

A final issue relating to integration was problems with version compatibilities between the software environments in which components were developed and those in which they had to be integrated. This was most apparent when migrating the DPE to the trial environment. While DPE development and testing went smoothly, installation on the trial environment at the UK national

host site produced a failure on one of the constituent components. Comparison of operating system and ORB versions between DPE development platform and the trial environment showed they were exactly the same. The problem was finally traced to a minor version difference between compilers that resulted in a difference in compilation. Similar problems occurred with other components, which would compile with no problems on developer's own systems, but not at the national host site.

5.5.2.2 Deployment Issues

The deployment exercise was again assumed to be a straightforward process, as it followed on from integration that should have identified and resolved all problems with the software. However, the assumption that component standards and tools will necessarily enable easy deployment may not be correct, as demonstrated in this case study.

In fact, in DOLMEN, deployment took twice as long as anticipated, and was still identifying problems with both the developed software and also the deployment environment. For example, the laptops chosen as mobile clients were found to be unable to cope with the load placed on them by the DOLMEN architecture, which required them to execute a number of multi-threaded processes all requiring system resources at the same time.

The deployment phase was also hampered by the fact that the same problems identified during integration testing were still being resolved. This meant that, for some scenarios, deployment was the first time they could be tested.

5.5.2.3 Trialing Issues

The final phase of the software-oriented aspects of the project was the architectural trial. In actual fact, problems throughout the development process had threatened the potential for a trial of any

kind. However, eventually the trial scenarios were executed, albeit, over schedule by a marked period. Salient features of the final trial report [70] were as follows:

The trial was broken into four areas, local trialing at national host sites (UK and Finland), simulated international trialing in Finland, and real international trialing between the UK and Finland over a dedicated broadband connection.

Results were broken down into an evaluation of functionality and an evaluation of performance. In terms of functionality, the following was concluded:

1. **Local Finnish trials:** In general the trial scenarios were executed effectively and repeatedly
2. **Local UK trials:** Due to problems interfacing with the UK mobile technology (GSM) and also problems with the developed software specific to the UK host site, trials had to be executed a great many times to obtain satisfactory results from all aspects of the service environment
3. **Simulated international trialing:** Due to problems in preparing the real international trial in time, a fallback position of carrying out the international scenarios at the Finnish national host site was adopted. International aspects of the trial were successfully demonstrated in this environment.
4. **International trial:** Problems with the availability of international communications equipment seriously affected the real international trialing. While low capacity signalling between the two sites was possible, the actual communication of data between the two sites, which required far greater capacity, was not possible. However, successful signalling between the two sites did demonstrate some international aspects of the service environment.

Performance evaluation was intended to assess the efficiency of the service environment and therefore the defined functionality of the DOLMEN architecture. Measurements were taken from the trials in the simulated international environment by adding logging capability to each component, via a macro that wrote a time and date stamp when any function was called. By examining the log files, timings for service machine functionality (for example, login, user registration, connection set up, etc.) could be calculated. For the information browsing application a comparison of the DOLMEN architecture against conventional web technologies should have been possible. A first year trial was run to characterise the performance of a traditional Internet architecture and also a mediated Internet architecture [123]. However, the measurements were not run to the same scenarios as the DOLMEN trials, so no useful conclusions could be drawn. In the case of the audio application no measurements for a comparable conferencing application were available.

While the actual performance measurements provide little information in the absence of a meaningful comparison, they do demonstrate the speed at which a component-based application can operate. In general, it was slower than might have been expected. For example, login took 1.5 seconds, while establishing an information browsing session was 28 seconds. While low capacity network connections can wreck distributed system timings, it does also appear that a contributing factor to the delays could be attributed to additional complexity of a component-based approach.

It is also interesting to note that measurements on different operating systems ORBs showed significant differences for similar functionality. This presumably demonstrates performance discrepancies between ORB implementations themselves, and also in how effectively the ORBs interact with the operating system.

5.5.3 Reviewing the Results and Goals of DOLMEN

The DOLMEN project received high praise from both peers and EC auditors. It was judged to have achieved both the trial and overall goals.

However, our concern is to examine the use of components in a large-scale software project and to assess the impact a component approach had on management and development techniques. Against these criteria, DOLMEN provides much food for thought. Hopefully, analysis of the many problems encountered may be useful to those working on similar projects in the future.

In following section we review the problems within the DOLMEN software development process and their resolution, consider which of them were either reduced or accentuated due to the component-based approach, and draws conclusions against the propositions defined in section 4.2.1.

5.5.4 DOLMEN as a Component-oriented Software Project

During the final stage of the DOLMEN development process, between integration testing and trial execution - a time when it was becoming increasingly apparent that there were major problems with the development of the trial software - the project manager published an internal report [157]. It discussed the chain of events leading up to the "trial crisis" and put forward reasons why this crisis may have occurred. These were as follows:

- "1. *complexity of the software under development,*
2. *instability of the CORBA run-time products when explored in their extreme features, as done in DOLMEN,*
3. *lack of adherence to Project-recommended software practices by some developers,*
4. *lack of mutual understanding between some developers of "neighbouring" modules,*

5. *definitely late delivery of some modules, which prevented testing in due time.*"

The following considers each of these points:

1. It is true that the software under development was complex. However, it was by no means on the extreme leading edge of software development – essentially the project combined well established telecommunications techniques (management interfaces to hardware) with new ideas (integrated environments to access said interfaces). The chosen implementation technique (CORBA), while relatively new, was based on a standard which had matured over a number of years, and was chosen to ease the development task, not complicate it.
2. The issue with CORBA implementations has been highlighted in industrial research (for example, see [132]) and certainly did not help in the development of the DOLMEN software. However, it was exacerbated by lateness in identifying these problems, during an integration phase, which made little provision for unexpected, time-consuming problems.
3. Project recommended software practices were introduced *during implementation* in an attempt to combat problems with software integration. For example, a software quality manager role was established as a single point of contact for the submission of software, and standards were established for documenting items delivered. However, by this time, while the majority of developers were attempting to adhere to project practices, they were under a great deal of pressure to deliver whatever had been developed in whatever form was available. Therefore, even with the best of intentions, it was very difficult to adhere to newly introduced practices.
4. The lack of inter-developer communication was discussed in earlier sections, and was certainly a major problem, resulting in problems throughout the development process.

5. Late delivery of software has also been highlighted as a factor hampering integration testing. However, here again, it was not until the integration phase that the issue was flagged, and by that time it was too late to employ countermeasures to combat the problem.

It would seem that the problems did not lie wholly in the technologies chosen for the project, or the requirements on the developed software, but in the management and selected process approach for development. The following are identified as the most problematic areas:

- **Rigid development process:** The rigidity of the development process, as illustrated in Figure 5-4, undoubtedly hampered the development of the software. The software was based on new technologies and unclear requirements – if the software was intended to validate the architecture, that architecture could hardly constitute an effective requirements definition. The project would have benefited greatly from systematic iteration and review at all phases of the development.

As well as the rigidity of approach, development also suffered due to the tight schedules for each phase. Effectively, while DOLMEN was a three and a half year project, implementation did not start until the final year (i.e. two and a half years into the project). This meant that implementation, testing, integration and deployment all had to take place within 12 months. The twelve months before implementation were spent on static and dynamic modelling, the effectiveness of which came seriously into question when the intended outputs of these phases (fixed IDLs and MSCs) did not emerge. A more effective approach would have been to incorporate a number of design/implementation/integration/review iterations over two years.

A final criticism of the chosen development approach is in the area of testing. Testing of individual components achieved very little, and wasted time that could have been better employed on integration testing and deployment

- **Lack of inter-developer and management-developer communication:** As we have seen, this point was acknowledged within the project and was certainly the source of many problems. The practice of dissemination of changes by the publication of reports announced via personal emails proved ineffective. Often, only some of the people concerned received notification. In other cases requirements were changed following management discussion, and again, these were not effectively communicated. Formal communication procedures (for example, via a single point of contact) for the release and modification of software documentation would have eliminated most of these problems.
- **Weak version control:** As evident in interface definitions, MSC specifications and implementations, the version control at all stages was virtually non-existent, resulting in different developers using different versions of the same thing (and because of poor communication, not being aware of any difference). If nothing else, the DOLMEN project highlights the need for version control at all stages of development (from requirements to implementation), not just at implementation.
- **Lack of understanding of the requirements and the technologies involved:** There was undoubtedly a lack of understanding of both the software requirements as a whole and also of the technologies used for implementation (i.e. a component approach). This was particularly apparent in:

- **SDL modelling:** The SDL approach used to identify inter-component interactions provided an effective simulation of the DOLMEN components, but a lack of understanding of the mapping between DOLMEN components and CORBA objects meant that the resulting model was of little direct help in the implementation of the software objects.
- **Inter-object dependencies:** The later phases of development suffered because developers were awaiting delivery of objects required to test their own objects. The identification of dependencies could and should have been a part of the requirement definition. Scheduling could have been more object specific and most of the problems could have been avoided.
- **Deployment:** The assumption that the component architecture would simplify deployment demonstrates a lack of understanding of its strengths and weaknesses. While component platforms provide the mechanisms to deploy distributed architectures, detailed knowledge of both these mechanisms and of project specifics are required for the deployment to be effective.
- **Mistaken assumptions:** Finally, the project suffered from a number of mistaken assumptions regarding aspects of implementation. This was particularly damaging because resultant problems were mostly not discovered until the integration and deployment stages, when there was too little time for contingency. For example, the discovery that the mobile terminals were not capable of hosting a large number of CORBA objects was not made until software was delivered, installed and about to be deployed. Again, the interoperability between C++ and Java versions of Orbix was also not discovered until integration. A more basic assumption was that using CORBA would make for swift software development. Consequently, the development schedule, for a project of this complexity, was extremely tight.

5.5.5 Learning from the DOLMEN Experiences

The following considers what the DOLMEN experience tells us about the management and development of future component-oriented projects.

5.5.5.1 Management Issues

- Linear development models are not appropriate for projects with significant novel aspects. The use of iteration and review is essential to refine requirements, designs and implementation.
- Change management is an essential part of a large-scale project. In the case of component-based projects, control of an interface definition is absolutely essential.
- Component dependencies should be identified at requirement definition and their impact upon the development schedule given due consideration. The components on which others are dependent should be scheduled ahead of those that offer no services to others.
- Formation of inter-dependent sub-groups of developers can be useful, but only if group findings are effectively communicated to the rest of the project team. This hierarchical approach came into being in DOLMEN informally in a number of areas, such as application and service session component development, communication and connectivity session developers and access and service session developers. However, in DOLMEN, change communication outside of these groups tended to be ineffective.

- Version control is essential for the software under development, and also for all development tools and operating systems involved.

5.5.5.2 Development Issues

- Interface definition provides the means of defining contracts between component server and component client developers. However, some form of functional specification is also required to clarify understanding of the functionality provided by the component. The use of MSCs, coupled with the use of interface definition, enabled clear understanding of the expectations of a given DOLMEN object, enabling parallel development.
- Techniques for defining inter-component interfaces and behaviour can be very valuable. Message Sequencing Charts were used to great effect in the DOLMEN project.
- SDL was too far removed from implementation to be of benefit. However, a greater understanding of the mapping between architecture and implementation may result in a more effective use of SDL for *some* domains.
- New technologies, such as software components, should be carefully assessed to see if they offer real advantages in meeting project requirements. Experience with DOLMEN was that:
 1. Component systems place a heavy load on hardware and software resources and are potentially slower to execute than “traditional” systems.
 2. Old hardware may not be enough to cope with the additional load a component approach may place upon it.
 3. Component standards do not necessarily guarantee “common” component functionality.

4. Compatibility testing should start long before implementation.

- Technology trialing at the feasibility stage is a good way to assess the functionality offered by a component architecture and the hardware and software platform on which the system will operate. It should be carried out as early as possible to ensure enough time is available to resolve problems.
- Testing software components in isolation is of limited value in the development of component-based systems with high levels of inter-dependency. It is virtually impossible to test functionality without dependent components. Communication between components is central to functionality and, therefore, it is necessary to test in an integrated environment to ensure correct behaviour.
- Deployment is an essential part of a component development process and is by no means a simple procedure. Component standards and tools, at the current time, offer little in the way of help. Effective deployment also requires detailed understanding of both the application and the component architecture.

5.5.5.3 The Impact of Component Techniques on DOLMEN

As a final part of the analysis of the DOLMEN experiences, we focus specifically on the use and impact of component technologies.

The component platform (see section 5.4) proved very effective in removing the need for low level programming to distribute the software platform, and also provided a great deal of developer support, via component services, for the complex mechanisms of mobile communication. Therefore, the greatest improvement in productivity was at a low level. While

this is not explicitly shown within the DOLMEN development process (the distributed environment was largely an assumed aspect), it is important to highlight it. Without the component architecture the software development task would certainly not have been achievable within the required time-scale.

However, as previous discussion illustrates, component technologies do not solve problems in the management of the software development process. Many of the issues in the DOLMEN project are common to all software projects when things start to go wrong. It is important to understand which were common software problems, which were specific to a component approach, and which were made worse by choosing a component approach. This understanding can aid in the adoption of component techniques in future software projects.

- **Late software delivery:** This is certainly not an issue peculiar to component-oriented projects; late delivery is one of the central themes of the software crisis [122]. However, one should consider whether a component-based approach adds complexity to the development, and therefore may impact on development time. While the development of software components, particularly when using a language such as C++ for implementation, does add some complexity compared to, for example, an OO project, the functionality encapsulated in the component architecture should offset this. Whether the overall effect is positive or negative depends on the nature of the project. In the case of DOLMEN, where a lot of low level functionality was encapsulated into the component architecture, it could be argued that the balance was achieved.

However, DOLMEN also suffered as it had no components from similar projects that could be reused. Within the ACTS framework there were other projects (for example, VITAL and

ReTINA) that were also developing TINA-based software components. As one of DOLMEN's roles was the examination of integrating mobile technologies into TINA environments, it was assumed that the primary focus of DOLMEN's development would be implementing mobile functionality. However, with no software provided from the other projects, all components for the DOLMEN architecture had to be developed "in-house". In theory, we could see a component approach benefiting the development productivity if other components had been available from other projects – if they were all developed to the same standard they would be compatible. However, DOLMEN demonstrated the problems with interoperability between CORBA objects. It is therefore uncertain how much benefit would have resulted from component reuse in this context.

- **Poor version control:** Version control is another topic not specific to component-oriented development. It is important in any large-scale development process where developer teams may be working on the same design or code. However, it could be argued that a component approach does introduce another aspect requiring very tight version control that is specific to component-orientation – interface definition. While interface definition can be seen as a way of defining functionality, and is therefore not very different to other development approaches, the nature of definition and the use of the interface for client development means that change to an interface may cause more problems than change to, for example, a static object model. If anything, then, component-orientation increases the requirement for effective version control.
- **Ineffective functional design:** At best we can see the ineffective functional design using SDL in the DOLMEN project as an experiment whose results were less useful than anticipated. At worst, we can view it as a failed technique that wasted six months of

developer time. As SDL has been used effectively in non component-oriented projects within the telecommunications domain, do we conclude that the failure was a consequence of component-oriented per se? Previous discussion has highlighted the difference between DOLMEN components and implemented objects and argues that this is possibly a reason why the SDL phase was of so little use. Thus, it is probably more realistic to conclude that SDL is not a suitable technique for component-oriented development, unless the mapping from architecture to implementation is simple, i.e., more or less one-to-one.

- **Problematic integration:** This is certainly not specific to component-oriented approaches. Indeed, in theory, the use of interfaces as contracts between client and server developers should reduce integration to simple component assembly. However, in DOLMEN the number of inter-component dependencies actually increased the problems of integration, especially when compounded with poor version control. Further discussion regarding the level of inter-component dependencies is included below. However, the problem was perhaps specific to the way DOLMEN implemented a component approach rather than something likely to surface in all such projects.
- **Problematic deployment:** There has already been discussion regarding the impact of a component approach on the deployment of a software system. Component-orientation certainly adds complexity to deployment unless the deployers are knowledgeable in both the component technologies and the organisation specific software.

5.6 Consideration of Findings Against Case Propositions

5.6.1 Proposition 1

Adopting and using component technologies in software development processes will affect process activities

This proposition is confirmed based upon numerous issues identified in the case review. In the DOLMEN project, the effect is generally to make the activity more complex. In particular, areas where component technologies are supposed, according the industry literature [33], to be most powerful – integration and deployment – have been greatly affected. Additionally, design activities were affected as previous techniques provided unsuitable for use with component technologies. Implementation activities were also affected due to problems with the chosen technologies.

However, while we can state that component technologies undoubtedly affect development activities, we should consider whether the problems that occurred with their use were as a direct result of component-orientation itself, or whether the rigidity of development approach was also a contributing factor. In an iterative model, such as Boehm's Spiral Model [19], development activities are placed in an loop that includes risk analysis and reviews. With iteration and risk analysis, could we expect problems that were unexpected in this case to have been identified and contingency measures are put in place? While it is not possible to re-run the case study with a different process approach, this consideration arising from this proposition identifies a need that could be further investigated in subsequent study.

5.6.2 Proposition 2

An awareness of the issues involved in the adoption and use of component technologies can ease their integration

The inverse of this proposition can be tested in this case study, as there was an assumption at the start of the project that component-orientation would solve a lot of development problems and that there was no need for special consideration of component based issues. The outcomes of this assumption can be seen throughout the project where unexpected problems have arisen. For example, the issue of interoperability between CORBA implementations has been documented to some extent in industrial literature. With an awareness of this issue, the project management could have ensured the use of common CORBA implementations across the project or, at least, run compatibility tests with implementations prior to integration testing.

Therefore, we can state that a lack of awareness of the issues involved in the adoption and use of component technologies can cause problems with their integration. Subsequent study can test the proposition in a positive way.

5.6.3 Proposition 3

Component technologies ease the development, integration and deployment of distributed systems

This proposition can be considered complementary to the first proposition in its statement of affect upon development activities. We can certainly consider the negative affect of component technologies upon development activities within this case study to conflict somewhat with the proposition. However, we should consider the issue of distribution in the proposition – did the component technologies contribute to a more effective development approach within a distributed environment? As discussed in section 5.5.5.3, one of the greatest gains in productivity that came

from the project was the distributed processing environment that let developers implement their component from a location independent viewpoint. Therefore, we can conclude that component technologies did contribute to easing the development of a distributed system. However, this benefit must be offset against the problems of deployment and integration that may have been as a result of the component technologies or, at least, as a result of a lack of knowledge in their use. Therefore, we must consider the proposition to be tested but inconclusive and, again, an issue that should promote further study.

5.6.4 Proposition 4

Uncontrolled adoption and use of component technologies can have a negative affect upon a development project

This proposition can be considered complementary to second proposition relating to the awareness of issues involved in using component-orientation. We can consider the adoption and use of component technologies to be uncontrolled in the DOLMEN case. While the review of available distributed platforms did serve as some kind of technology assessment, albeit centred wholly around literature review, there was no transfer strategy that followed this evaluation. The issues identified above certainly illustrate aspects of negativity that have resulted from the uncontrolled transfer. Once again, the degree of negativity may have a contribution from the development approach and would benefit from further study. However, we can state that in this case, the proposition has been demonstrated to be true.

5.7 Chapter Summary

The DOLMEN case study provides a large scale industrial example of the use of a specific component technology to address a software requirement in a specific domain. Additionally, it

has been very effective in highlighting problems with the use of new software technologies in a development project.

We cannot consider conclusions from this case study to be indicative of all use of component technologies within development projects. As discussed in chapter 4, it is hard to generalise from a single case study: it is difficult to identify common issues and those that are as a result of uncontrolled factors within the case study. However, by considering case findings against case propositions, we are able to develop theories regarding their use that can be tested in subsequent study.

The following chapter presents the second case study in the research programme – related to the use of component technologies within an Independent Software Vendor specialising in network management solutions. The use of two case studies allows the examination of component technologies in two separate contexts. This second case provides an opportunity to test general case propositions in a different context, to test theory developed from propositions in the first case study, and also to test new propositions. All of these can contribute further to the development of theories regarding the adoption and use of component technologies while increasing external validity of findings.

This chapter presents the second case study, and is similar in structure the first. While this case study is quite different, a comparison of the effects of component-orientation in each case enables a focus of theories in the adoption and use of component technologies.

6. The Use of Components in the Network Management Domain

6.1 An Overview of Netscient Ltd.

Netscient Ltd. is a SME specialising in the development of network planning and design systems for communications service operators and providers. It was formed from the planning team of AT&T Unisource in 1998, applying knowledge developed planning and designing company specific networks to the wider network management domain. Their motivation for becoming an Independent Software Vendor (ISV) was the realisation that the planning and design process is essentially the same whatever the details of the particular network under consideration.

The case study follows Netscient software development practice in its first year of trading. It provides an important contribution to the present research as, for an ISV, the start up and development of a software infrastructure is crucial in delivering products on time and on budget (to meet company schedules and to please backers). The use of a number of techniques, including components, did result in the successful delivery and enabled this SME to compete with far larger software houses in the production of quality software.

This chapter reviews the Netscient organisational structure, and the domain in which it exists, before focussing on the company's approach to developing software. This includes domain analysis, choice and use of software technologies in the development process and the nature of the software development process itself. Use of the technologies within Netscient is illustrated by discussion of in-house systems for the management of network equipment personalities (see

section 6.4.1.3). The case study centres on this aspect, as this was the area that had made most use of component technologies. However, the other uses of component technologies within the organisation are also discussed where appropriate.

6.1.1 Netscient Organisational Structure

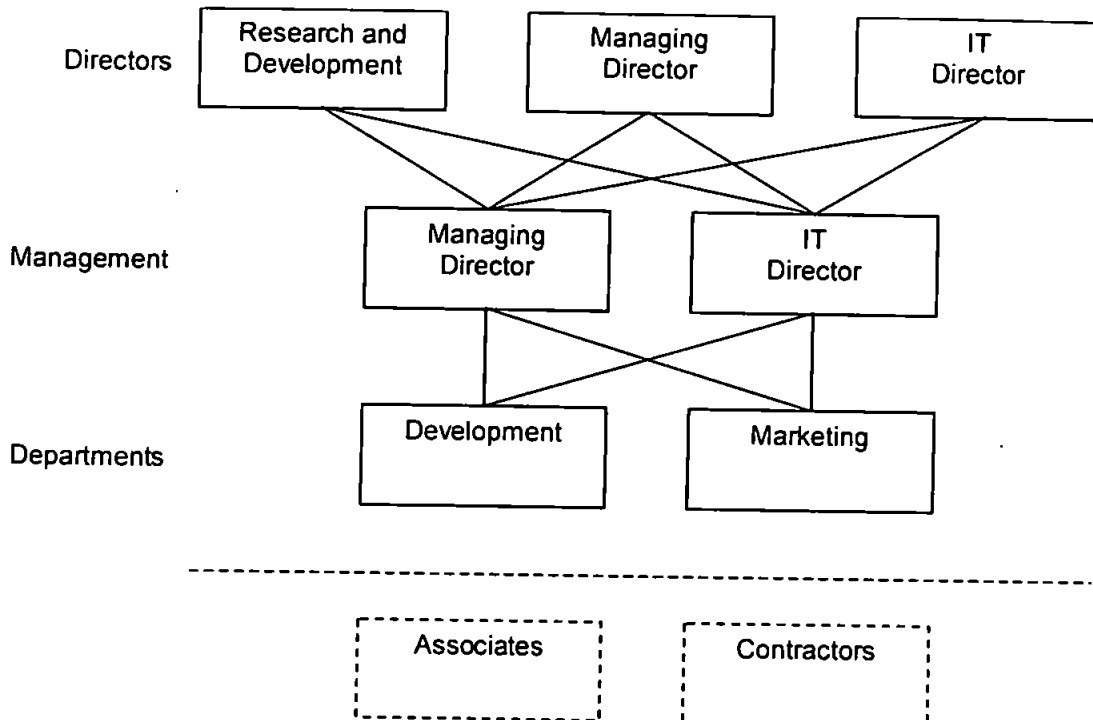


Figure 6-1 - Netscient Organisational Structure

As one would expect with an SME, the Netscient organisation structure is fairly simple. While there are formal distinctions between directors, management and department personnel, communication between layers is relatively informal – directors are as likely to communicate directly to department personnel as managers. Additionally, Netscient deal with both associates and contractors. Contractors provide specialist knowledge in order to perform tasks within the development process, which are beyond the skills of the core development team. Associates work in a consultancy capacity advising on direction, and introducing new skills and techniques to the organisation. The case study was made possible via such a position: while Netscient served as a

case study in the use of component technologies, as an organisation they received consultancy regarding the use of leading edge software techniques.

6.1.2 Product vs. Domain Orientation

An early decision by the directors of Netscient was to take on a software development strategy from a domain oriented viewpoint. The following briefly differentiates between a product- and domain-oriented view to introduce the topic in the context of this case study.

- **Product-oriented:** Requirements and objects are identified on the basis of what is required for a specific product. Requirement analysis results in the identification of objects that interact to provide application functionality. Object definition develops the behaviour and data specific to those defined classes.

This approach is illustrated well in the DOLMEN project. The initial requirements definition for the project was architectural, defining the DOLMEN *product* – the DOLMEN Service Architecture [158]. Following overall architectural definition, the focus moved toward the objects required to achieve the architectural functionality. This object definition drew from other projects (Research and development in Advanced Communications technologies in Europe (RACE) projects [124], TINA) that also addressed service architecture functionality. Component definition provided a catalogue of components that would be developed to achieve the DOLMEN Service Architecture functionality. However, while one of the initial aims of DOLMEN was to produce a set of TINA-compliant reusable components, the defined DOLMEN components were focussed solely on implementing the DOLMEN product. Therefore, there was little reuse potential for the components outside of the DOLMEN environment. Admittedly, the components would be reusable in architectures based on the

one defined by DOLMEN, but even integration into another TINA compliant environment would be a complex task.

The main advantage of a product-oriented approach is that it requires far less planning and analysis than domain-orientation. The drawback is that, as demonstrated by DOLMEN, the reuse potential for the objects and components is low.

- **Domain-orientation:** In a domain-oriented approach, initial analysis is not based around the required functionality of a product, but focuses on what processes and actors exist within the organisational domain. The theory is that software developed for domains (whether they are horizontal or vertical) use similar components through different applications. Such domain encapsulation is well illustrated by office suites (e.g. Microsoft Office, Lotus SmartSuite). At a coarse level of granularity, all provide the same components (word processing, information organiser, spreadsheet, and databases). At a finer level, similar functionality is required within the applications themselves. For example, word processing, organisers and even spreadsheet applications require spell checking. Graphing and data presentation is required in spreadsheet and databases - the embedding of such functionality inside a word processor is also desirable. The Microsoft Office suite is perhaps the most effective example of this domain orientation. Through each progressive release of the suite, more and more functionality has been encapsulated in common components and accessed via the COM standard.

A domain-oriented approach's primary drawback is the time it takes to carry out analysis and development work. It also requires a great deal of domain knowledge and experience in order that it be successful. The theoretical advantage of a domain-oriented approach is that the

reuse potential is far higher than a product-oriented approach. It was hoped that the Netscient case study would help confirm or disprove this contention.

6.2 The Netscient Case Study

The Netscient study was the second case used in the assessment of the effect of component technologies upon software development. It provided a different context to examine the ways in which component technologies could be used, specific differences being:

- A different vertical domain, but one that was not so far removed from DOLMEN as to be entirely incomparable (for proposition 4 – see section 4.2.1.3);
- A domain, rather than, product focussed approach to development;
- A more cautious directorial view of component-orientation – it was not regarded from the outset as *the* technique for software development;
- Use of a different set of component technologies – Microsoft COM-based technologies rather than CORBA-based.

Therefore, the case could help in identifying common issues in the adoption and use of these new techniques. It should be reiterated that the Netscient case study does not complement the DOLMEN case study as part of a multiple case study approach with matching propositions. It is a single case study assessing the effect of component technologies upon software development, its propositions guided somewhat by the theories developed from the DOLMEN case.

6.3 Case Study Definition

Chapter 4 has discussed the general approach to the case study and the research methods used. This section examines issues specific to the Netscient case, based upon the discussion above regarding the value of the study. Firstly, it reviews case study propositions before elaborating

upon the analysis approach, discussing strategy and types of evidence used. Finally, it defines the structure for the case study report, which makes up the remainder of this chapter.

6.3.1 Case Study Propositions

The propositions for the Netscient case study comprises both general case propositions and Netscient specific propositions, defined in section 4.2.1.3, and repeated below:

1. Adopting and using component technologies in software development processes will affect process activities
2. An awareness of the issues involved in the adoption and use of component technologies can ease their integration
3. A domain-oriented approach to component development provides a greater degree of reuse than a product oriented view.
4. Similar issues with component-orientation occur when using different technologies from the same field (i.e. Microsoft based, rather than OMG based technologies)
5. Issues in the DOLMEN case study can be avoided through greater knowledge of the technologies involved

6.3.2 Case Study Role

As with the DOLMEN case, the author played a participative role within the study as a result of funding for the research programme coming from the Netscient organisation. Again, this role was at a development level, in this case focusing upon the development of in-house systems (discussed in section 6.6.1). Development of the in-house system was carried out by three developers in all, reporting back to the IT director for development strategy.

The development role enabled access to developers on a peer level with the benefits that brought to data collection, and provided the opportunity for participant observation within the case

context. Additionally, an associate role within the organisation provided the opportunity to provide directors with opinion regarding technological issues (for example, the suitability of a certain development tool or technique). This associate role provided the opportunity to feed in “lessons learned” from DOLMEN in an advisory capacity. The directors could then have a more informed decision in their selection of technologies, enabling the testing of the proposition that questions whether an awareness of issues in component-orientation mean a more effective use of them. The role did not, however, have any contribution to strategic direction or have any direct control over the specific approaches chosen by the directors or in the management of any of the development projects. In this role, the effect of component-orientation was assessed through direct observation backed up with documentary evidence and interview.

6.3.3 Analysis approach

The analytical approach was similar to that of the DOLMEN case study (see section 5.2.4) – it centring on explanation building and the development of theories in the adoption and use of component-orientation. Evidence types were also similar to those of DOLMEN:

- **Participant observation** – in a role of analyst/developer for in-house systems, hands-on experience with the use of component technologies could be obtained. Additionally, liaison with the product development team leader provided the opportunity to informally discuss issues related to the development technologies.
- **Direct observation** – evidence was collected in a similar way as participant observation, which generally came from project management and strategic issues, or from aspects of development in which participation was not possible.

- **Interview with project personnel** – in person, via email, and also through telephone conversation.
- **Documentation** – while no formal deliverable documents were specified in the Netscient case, several types of documentation were available
 - Internal working papers
 - Meeting notes
 - Project email

Examples of evidence used in the analysis of this case study are included in appendix C.

6.3.4 Case Study Review

1. **The Netscient Software Development Process** – reviewing the Netscient software development process, a point of reference when assessing the effect component technologies had upon software development in the project.
2. **The Netscient Software Platform** – defining the mix of software technologies used within Netscient as their platform for software development. This is a useful point of reference in understanding the case outcomes.
3. **Case Study Analysis** – identifying issues arising from the development of software within Netscient, focussing upon in-house systems, and analysing the issues identified. Consideration is made to possible causes for each issue based upon case study evidence. and also consideration of the issues against case propositions.
4. **Case Study Results** – developing the issues identified from the development review against case study propositions. The results in this case also consider unexpected outcomes that have emerged from analysis of the use of component technologies in this case.

6.4 The Netscient Software Development Process

In general, the Netscient development process was less rigid than that of DOLMEN (see section 5.3). In considering the development process, two aspects are the most important:

- **Design standards:** Discussion of organisational structure emphasised the distributed nature of Netscient development. Activities often progressed in parallel, so that developers, or developers and contractors, were working from the same designs on different aspects of an application or on different applications. Therefore, as in DOLMEN, design standards and application interfaces were central to specify object models, etc., that could be clearly understood by different developers. For object and component models, the Universal Modelling Language (UML) [58] was used. At this early stage, the only application interfaces were for the communication of personality details between in-house and product applications. XML Document Type Definitions (DTDs) provided a straightforward method of specifying them.
- **Ongoing review:** Throughout design and development, review and iteration were effective in ensuring that everyone was still working toward the same goals, and requirement definitions were being met. Directors' experience of the domain was extremely useful as they could act as reviewers with a good understanding of what the user would expect. In the event that a review phase resulted in alterations to design or interfaces, design updates were communicated to all personnel for review and integration into the development process.

Figure 6-2 illustrates the development process:

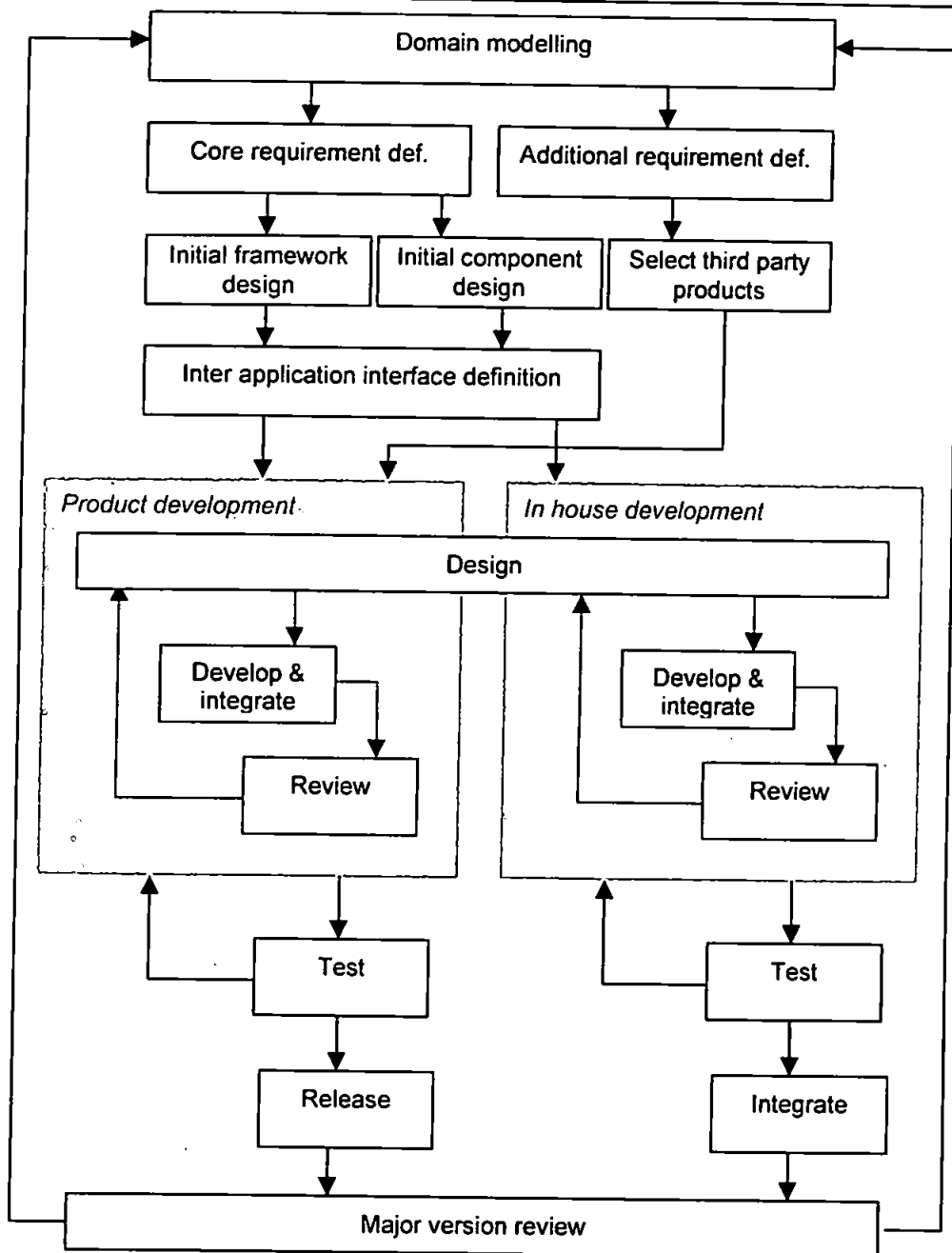


Figure 6-2 - The Netscient Software Development Process

Most of the activities in Figure 6-2 have been discussed already:

- **Domain modelling:** See section 6.4.1.

- **Core requirements definition:** Basically two major tasks. Firstly, defining requirements from the network planning and design business process – to be implemented in Netscient's object framework - and secondly, defining requirements for personality administration – to be implemented as an in-house system
- **Additional requirements definition:** These were summarised in Figure 6-4.
- **Initial framework design:** Production of an initial object hierarchy to encapsulate core business activities.
- **Initial component design:** Production of component diagrams for the definition of in-house components
- **Select third party products:** Determine which third party products will meet additional requirements
- **Inter-application interface definition:** Define information interfaces between personality systems and product software
- **Product development:** The process of developing software product applications..
- **In-house development:** The process of developing the personality management infrastructure (see section 6.4.1.3).

- **Design:** As overall design impacts on both in-house and product software, the design phase spans both processes.
- **Develop and integrate:** The development of domain components (either object framework or personality component classes), followed by applications incorporating them along with third-party components.
- **Review:** At set points in the development (completion of class definitions, implementation of core functionality, database interfacing, etc.) reviews assessed the course of the development and determined whether any modification to design and direction were necessary.
- **Testing:** Standard testing to assess functionality against requirements. Test findings sometimes resulted in a feedback to the development activity.
- **Release/integrate:** In the case of product software, a version release was carried out following full testing. For the in-house personality system, integration into work practices followed testing.
- **Major version review:** Following release and integration, major reviews took place to assess next version functionality, lessons learned from the previous version release, additional requirements, etc. These then fed back to the domain model, restarting the whole development process.

6.4.1 Netscient Domain Modelling

Netscient adopted the domain-oriented approach for product software and in-house applications. The following discusses the method used by the organisation in identifying their domain and defining domain objects.

6.4.1.1 Core Requirement Analysis

The start of the domain modelling process was to consider the nature of the domain in which the company exists – namely network planning and design. The requirements analysis must identify core business processes, and the objects required to perform the necessary transformations in those processes. Such modelling requires a high degree of understanding of the domain, and it is significant that the directors of Netscient had more the thirty years relevant experience between them.

6.4.1.1.1 Process Analysis

The greatest pressure for communication network providers is meeting customer requirements for greater capacity, better quality of service or more connections. In order to meet these needs, managers must be able to assess the feasibility of a change on the network, plan how it can be carried out, and then implement it. The network may be highly distributed geographically, made up of numerous sub-networks comprising different equipment and management interfaces any change many affect a large and heterogeneous set of equipment. Therefore it is essential all changes are thoroughly considered and effectively planned before execution.

In the deployment of new networks, the problem for managers is much the same – how best to design the network to get maximum efficiency out of the equipment while fulfilling customer

requirements for connections and capacity. Again, careful planning and design (and perhaps simulation) are required to make the implementation phase as straightforward as possible.

6.4.1.1.2 Process Definition

The core business process for network managers is transforming customer requirements into network changes. There are three primary activities within this process:

- **Planning:** Formulating a long term view, anticipating the state of the network 6 to 12 months ahead of detailed design, based on current network growth, customer requirements, etc.
- **Design / scheduling:** Transforming high level requirements from the long-term view into the appropriate network infrastructure. Mapping high level requirements to specific equipment interconnections, and determining optimum routes, etc. for such connections. Design should also determine the order in which the implementation will take place – when dealing with live networks, it is not possible to take the whole network down for several hours while engineers implement network changes.
- **Deployment/delivery:** Network implementation and support will make changes based on scheduling information.

Another important activity, namely *control and administration*, drives the continual process iteration. The live network is analysed for performance, traffic profiles, capacity, etc., and this information is fed back into planning, where estimates of greatest loads, optimum routes, etc. help determine the future composition of the network.

The cyclical nature of network management core process is illustrated in Figure 6-3, taken from Netscient's website (<http://www.netscient.com>).

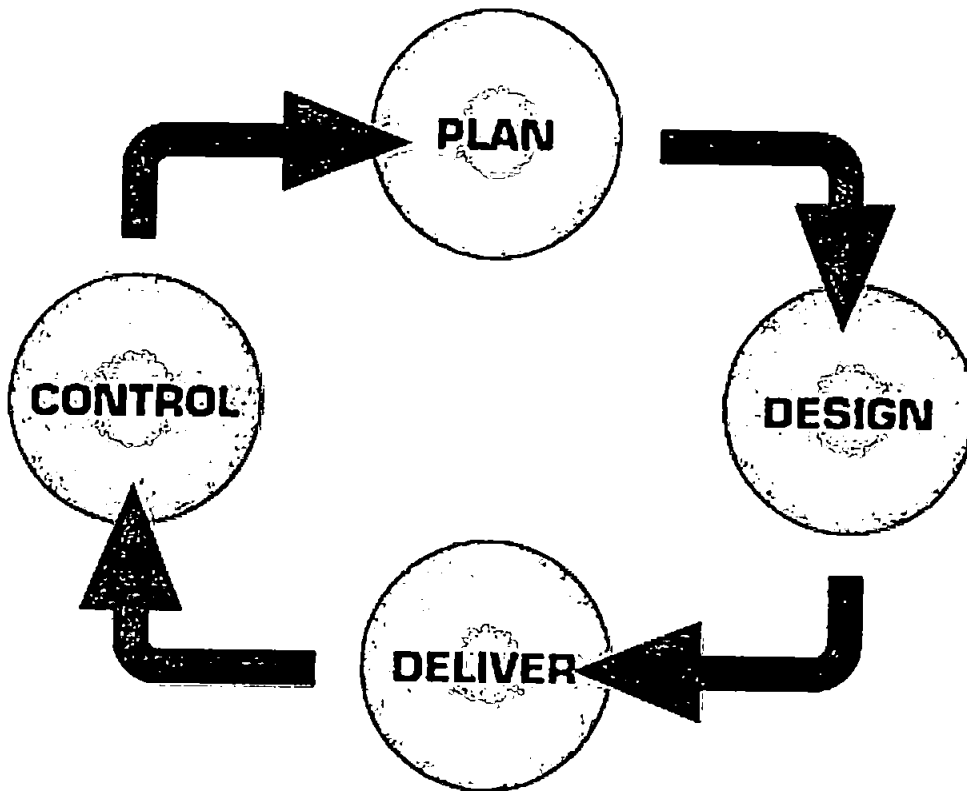


Figure 6-3 – The Core Network Planning and Design Process

6.4.1.1.3 Additional Functional Requirements

Alongside core processes for network management, additional functionality is required in network management applications. These areas were identified as:

- **Geographical Information Systems (GIS):** By their nature many networks are distributed over large geographical areas. Therefore, the most effective way of visualising current and future designs is through the overlaying of network plans onto geographical maps – the sort of functionality provided by GIS applications.

- **Scheduling / report generation:** The means to convert a network plan into a project and within that project to identify tasks within the overall schedule. Additionally, functionality is needed to manage customer orders and map them onto projects. Finally, there is a requirement to extract information from the planning and design system into clear, well-presented reports.
- **Graphing / diagramming:** Scheduling implies a requirement for workflow diagrams, project management charts, etc., In addition, diagramming functionality is also important for the visualisation of the network, by means of structured diagrams, etc.

6.4.1.2 System Object Analysis

Following process identification and market analysis, the domain model was developed to identify the objects within the system that are transformed and affected by the information throughout the business process.

To take a simple example, a customer of a cable company places a request to have a connection to their house. A member of sales staff takes the customer request and generates an order to introduce it into the system - the order formally defines the customer request. The order is then put into the management system. The manager determines whether a new physical connection is required and if so, where on the switch this connection can be made (card, port, etc.). Once planned, the system generates scheduling information for the project, identifying jobs that will need to be carried out in order to implement the change.

Even from this simple example, four sets of objects can be identified:

- **Network equipment:** Switches, nodes, cards, etc.

- **Network connections:** Trunks, circuits, virtual circuits, etc.
- **Project objects:** Projects, orders and jobs.
- **Human interfaces:** Customers, sales staff, managers, etc.

Of these object groups, all but the last fall inside the system. The human interfaces determine the system boundary as they act on the system, but exist outside of it. These are not modelled as part of the system, but are users of the applications developed within the Netscient domain.

6.4.1.3 In-house Requirements

Domain analysis was also undertaken for in-house functions, which would affect Netscient's own ability to provide effective software solutions. The main focus of this work was the definition and administration of *network equipment personalities*.

The concept of network personalities is a novel aspect of the approach used by Netscient in producing vendor-independent software systems. When considering the behaviour of a given piece of network equipment (for example a switch), while the base behaviour is always much the same (it comprises shelves which hold cards that provide the switch's functionality, power supply, means of connection, etc.,) there are also vendor specific aspects to each piece of equipment. These can be as simple as what sorts of cards are allowed in a shelf or more complex, such as software versions between compatible cards. It is to accommodate these differences that the majority of network management systems are vendor specific.

The view taken by Netscient is that most of what can be done with planning and design systems is generic – the manager assesses connections between switches, tests the feasibility of introducing a new connection to a switch, moves connections, determines optimum routes, etc. Therefore, if it were possible to take account of vendor specific characteristics outside the core applications, the

potential for reuse would be greatly increased. The necessary representation of a piece of equipment (or a connection or a project object) is referred to as an *equipment personality*.

Therefore, any customer, from any customer group, can use the company's software to model their network knowing that vendor specific limitations and behaviour would be handled via network personalities, supplied by Netscient to match their particular equipment. If a customer introduces new equipment, Netscient's support services can provide personalities for the new equipment that "plugs in" to the installed planning and design systems.

In-house requirements centred around the definition, storage and distribution of these equipment personalities. There was also a need for a flexible common interface between personalities and product software, such that new personalities could be dynamically added to the planning systems without having to release new system versions.

6.4.1.4 Definition of Domain Functionality

Figure 6-4 illustrates the functionality required in the Netscient domain. It shows core network planning and design functionality, together with the various additional areas of functionality which were described above.

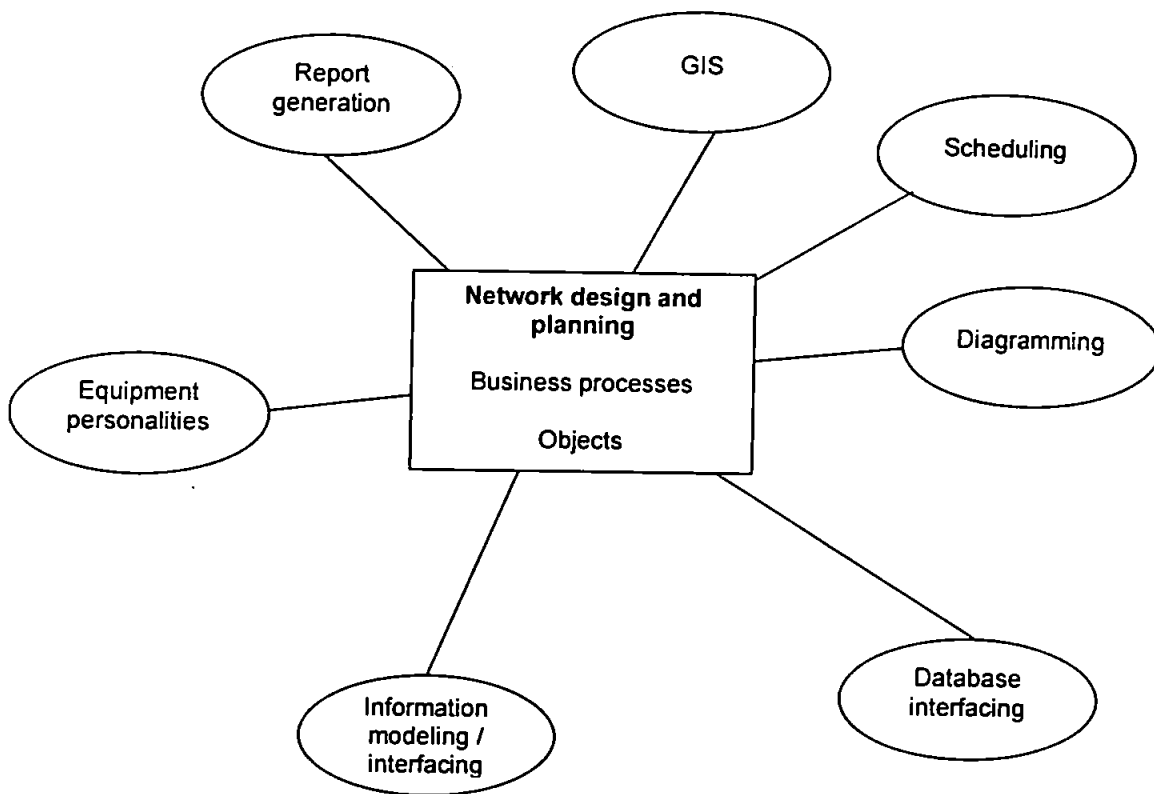


Figure 6-4 – Definition of Netscient Domain Functionality

6.5 The Netscient Software Platform

As with the DOLMEN project, component technologies were seen as an enabling technology. However, Netscient did not use components in all of its software development. It was noted very early in requirements definition that component technology was still a relatively unstable area – new standards and products were continually emerging and the major component vendors' primary aim seemed to be arguing why their approach was better than that of their competitors. Moreover, it was felt that component approaches brought a level of complexity that could not be justified for some aspects of development.

The main area in which component techniques were used are:

- **In-house systems:** Described in more detail in section 6.6.1, the in-house systems were used to address the administrative problems of managing vendor equipment in a generic way. However, as the development only affected internal processes, it was also used as an area to assess the use of component technologies without impacting upon development schedules.
- **Interfacing with third party functionality:** Domain modelling had identified a number of different areas of functionality required to enable the most effective planning and design solutions. As an SME it was considered far more appropriate to concentrate their own development efforts on encapsulating their core domain and buy in components that provide functionality for auxiliary domains.
- **Product customisation:** A service intended to be offered by Netscient additionally to its software products is the development of customer specific solutions, further integrating Netscient applications into the overall customer network management structure (for example, directly interfacing planning systems to management systems in order to automate some management functions). Component standards provide common interfaces between systems and, therefore, a component-oriented approach was considered appropriate for this area. Note that as product customisation is a feature that Netscient plans to introduce as a service in the future, it does not feature in the current Netscient Software Platform (see below).

The remaining aspects of system were defined using other, more mature technologies – primarily object-oriented techniques. Thus, for example, the internal representation of core equipment makes use of an object framework rather than component classes.

The resultant Netscient Software Platform accordingly features component technologies, object frameworks and information interfaces (see description below):

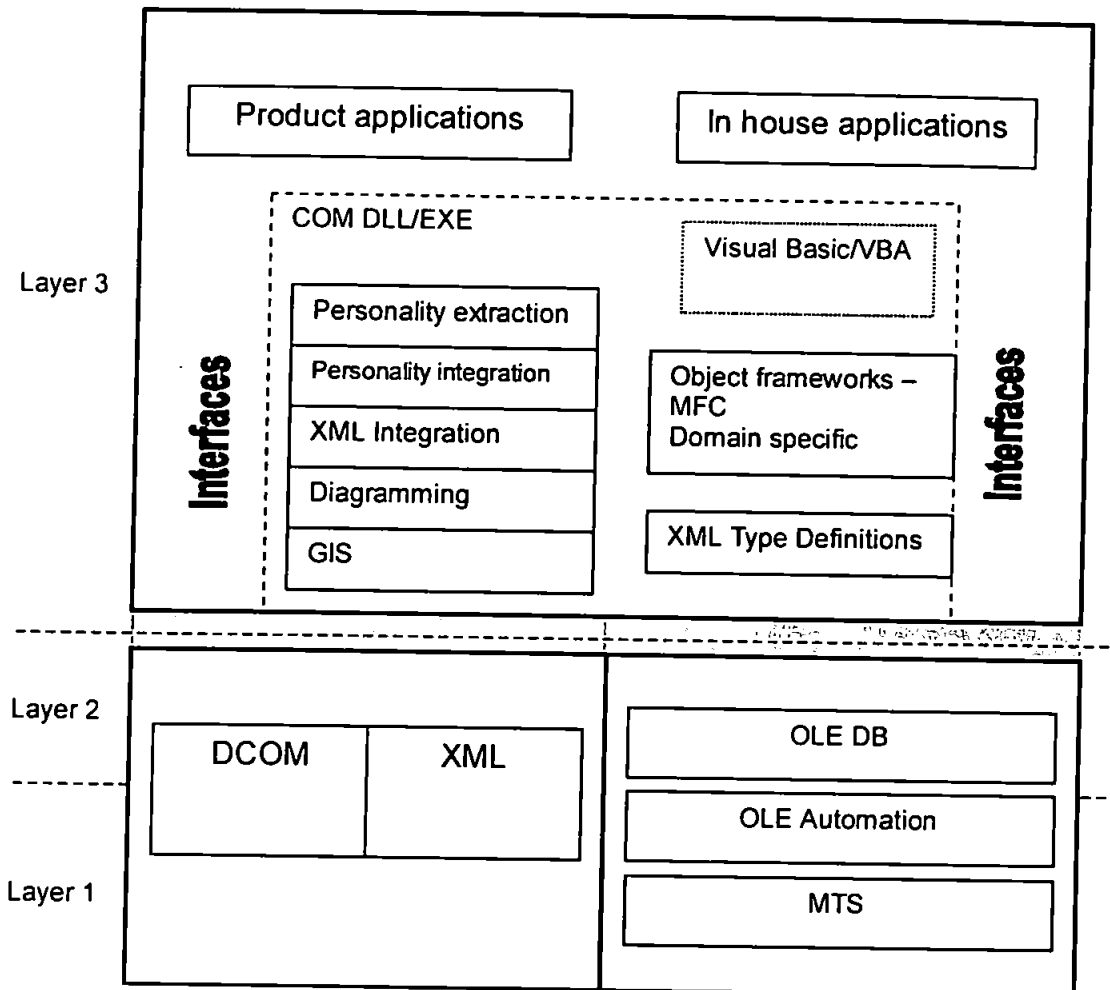


Figure 6-5 - The Netscient Software Platform

As with DOLMEN, the platform relates to the reference model for component platforms (see section 8.2). However, it is also divided into three technology areas:

- **Object frameworks:** Object frameworks use object-oriented techniques to encapsulate the functionality of a given domain into an object hierarchy, from which application specific behaviour can then be inherited.

- **Component technologies:** Those aspects of the software architecture based around component standards and services.
- **XML technologies:** Used to pass information (equipment personalities) between in-house and product software in a standard way, enabling information extracted from in-house administrative applications to be dynamically loaded into product software. This aspect is discussed in far greater detail in section 6.6.3.2.

Some salient points concerning the various levels of the architecture are as follows:

- **Standards:**
 - **Component technologies**

DCOM: As all developed software is for Windows platforms, DCOM was adopted as the core component technology.

- **Information interfaces**

XML: Information interfaces required a standard for the structuring of information. While database formats (e.g. storing as an Access or Oracle database) would have been possible, this would place a reliance on the use of a RDBMS for distributing personality information. While this is entirely feasible, it would add unwanted complexity to the applications. A more elegant approach is provided by XML [164]. Using XML, an application needs only incorporate a parser to be able to handle

structured information, regardless of the underlying network technology and the location of the data.

- **Services:**

- **Component technologies⁸**

MTS: Currently, MTS is used in-house to manage the communication between administrative clients and the database backend that holds the personality information (see section 6.6.1). This serves to demonstrate and prove functionality without having to impact upon software products. If this test goes well, Netscient expect to use MTS as the vehicle to develop more distributed Internet based products.

OLE Automation: The exploitation of third party products is another area in which component technologies are used. OLE automation is currently used to interface Netscient products with Seagate Crystal Reports, in order to add reporting functionality to the applications.

OLE Database: OLE DB is a service based on COM to provide uniform interfaces to diverse information sources (for example, email, groupware, RDBMS, object databases). The concept is similar to ODBC in which storage technology vendors develop their own interface implementations to enable a client to access each storage medium. Currently OLE DB is used to interface Netscient applications to RDBMS

⁸ A discussion of the COM technologies used in this section can be found in [36]

backends. In the longer term, using OLE DB should result in less effort in integrating other information resources.

- **Netscient Components:**

- **Object frameworks**

Netscient framework: The objects that encapsulate the Netscient domain (see section 6.4.1) in C++.

Microsoft Foundation Classes (MFC): The standard Microsoft C++ library for developing Windows applications [83]

- **Component technologies**

GIS: GIS functionality is achieved through the use of an ActiveX control developed by GeoConcept (see www.geoconcept.com) who also provide the associated geographic database.

Diagramming: Another third party component – Laselle Technologies AddFlow ActiveX control (<http://www.laselle.com>) - enables diagramming functionality to be incorporated into Netscient applications.

Database access: As a lot of RDBMS vendors do not yet support OLE-DB, it was necessary to incorporate some ODBC data access into the architecture to enable

“traditional” database interfacing. Microsoft’s ActiveX Data Objects library is used to this purpose.

XML integration: Product software needs XML parsing functionality to handle structured inter application information. The Microsoft Internet Explorer 4 XML parser provided this.

Personality administration: Locally written to interface with the equipment personality database backend (see section 6.6.1), and also obtain information regarding personality types, etc.

Personality extraction: The Netscient software provides the functionality to retrieve personality information and recast it in XML format.

- **Information interfaces**

Netscient Type Definitions: The type definitions define the information interfaces by specifying types and type structures for the generated XML files (see section 6.6.1 for more detail).

- **Clients:** make varying demands on the software infrastructure:

Software products: End-user products exploit the full power of the software architectures.

In-house clients: These are currently restricted to providing user interfaces for personality administration and extraction (see section 6.6.1) and do not use the object frameworks.

6.6 Case Study Analysis

As an examination of an aspect of development that used component technologies the in house personality management system is considered. This was also the area in which the researcher had the most participative involvement, and therefore the greatest potential to assess the effect of component technologies first hand.

6.6.1 In-house Personality Management

Figure 6-6 illustrates the use of components within the Netscient organisation, by detailing the application structure for in-house, personality administration.

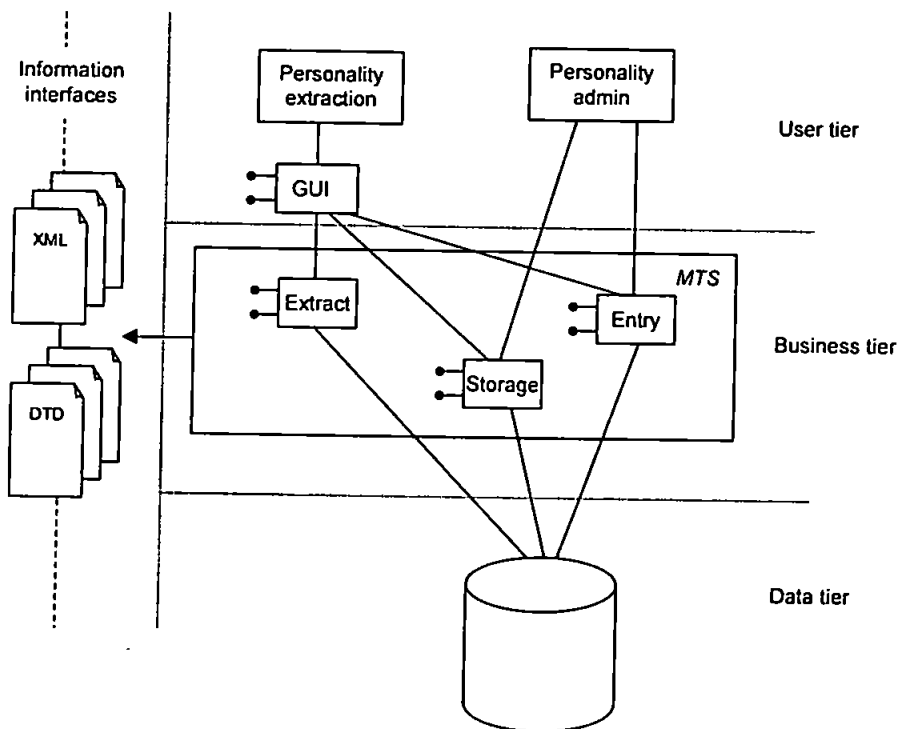


Figure 6-6 – Netscient In-house Application Structure

6.6.1.1 System Overview

The software is organised as a simple three-tier structure. The role of each tier is as follows:

- **User tier:** There are currently two clients, one which enables the user to browse and add to the personality store, and one to extract from the store and generate XML files. Figure 6-7 and Figure 6-8 provide a screenshot of each client.

The browser client provides functionality to browse all equipment types, edit existing entries and add new ones. This is implemented as a simple interface to the entry and storage component classes in order to obtain information about equipment types from the components and pass modified or new information to them.

The extraction client provides a list of each equipment type, so that a user can make selections to create an *equipment profile* (a customised collection of equipment mapping to a customer's specific requirements) and generate the required XML. Note from Figure 6-6 that the extractor client makes use of a custom GUI component. This implements an equipment type listbox, interfacing with the entry and storage component classes to obtain a list of all equipment entries of a given type.

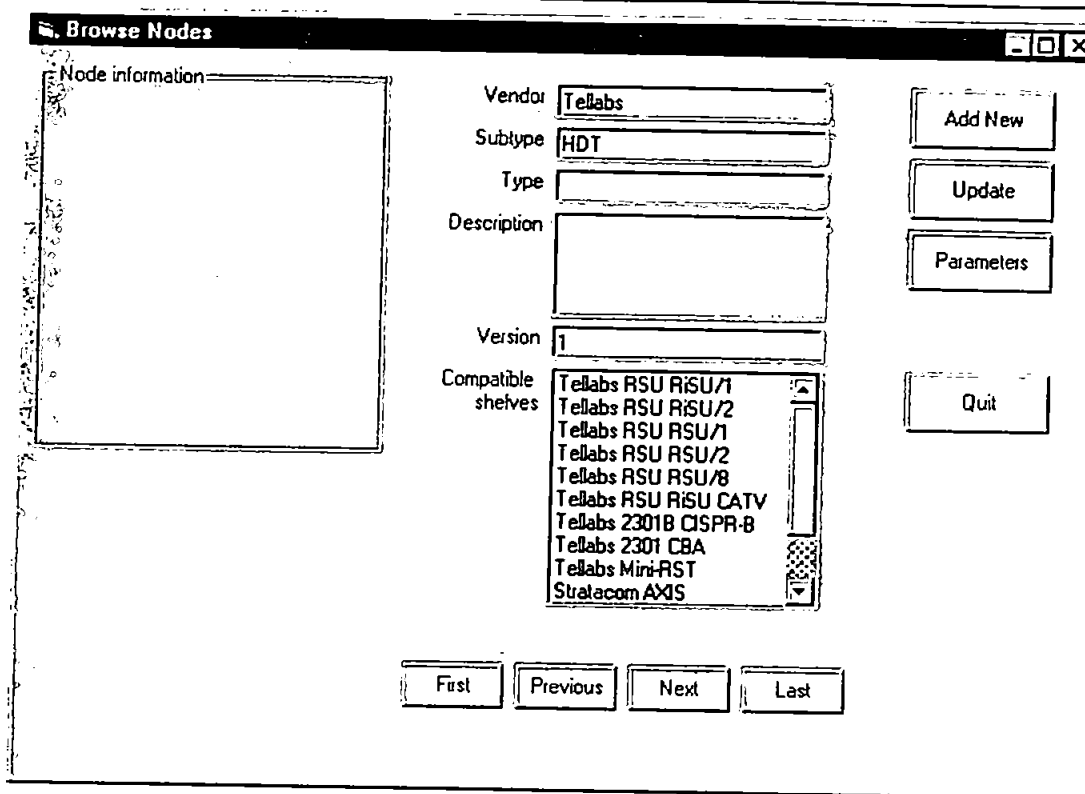


Figure 6-7 – Netscient Personality Browser

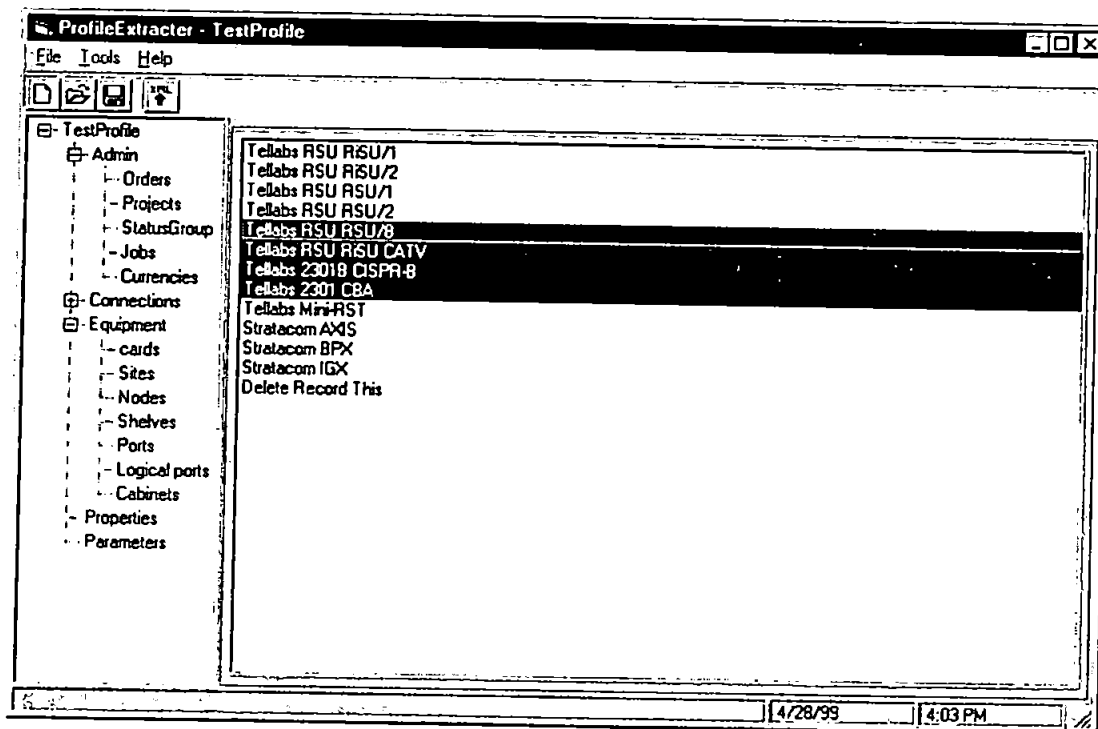


Figure 6-8 – Netscient Personality Extractor

- **Business tier:** The business tier defines a number of interfaces for equipment browsing and entry, a single interface for equipment extraction, and a storage class discussed below. Each interface set is actually implemented by two classes, supporting different levels of equipment detail (a base and detailed level). As each class provides an implementation of the same interface, clients can dynamically resolve the level of detail required and dynamically switch between them.

The storage class provides an ADT to hold basic equipment details (id and description) which can be passed between classes and clients so that different clients can access a given equipment definition via its id.

- **Data tier:** The database backend provides structured storage for the personality information. It is implemented as a simple relational structure within an RDBMS.

6.6.1.2 System Design

The design of the administrative structure served two purposes: firstly, to ease implementation of the system, and second, more importantly, to communicate the design to product developers in a structured fashion. This was essential for the information interfaces, which are the primary overlap between in-house and product software, but it was also important to demonstrate the overall system structure.

The design used simple, but powerful, techniques to express the structure:

- **Functional interface definition:** It was important to resolve what functionality each component class would offer before implementation. This ensured version control on each interface would keep *binary compatibility*. This means that while the method of

implementation can vary, the function definition (as defined in the interface – function name, in parameters, out parameters, return types) has to remain constant. Therefore, modification to the component implementation will not result in clients having to alter their code. Using the COM standard and Visual Basic, a developer can force a component to maintain binary compatibility. One possibility is to define the interfaces in Microsoft Interface Definition Language (MIDL)– the standard interface definition language for COM classes. However, it is easier to define them as simple classes in Visual Basic. The development environment then generates a *type library* containing the MIDL versions, which is used by the component standard for the component calls.

- **Information interface definition:** The information interfaces were defined using XML Document Type Definitions (DTDs). While recent development in XML enable type definitions to be written using XML, the parser used in the product software did not have this capability. Using DTDs, rather than the newer *XML-Schema*, provided the most portable way of defining the information interfaces. These information interfaces were a central element of the design, impacting on:
 1. **Database design:** The tables were defined to map to each defined element and its attributes.
 2. **Extraction component implementation:** Used to ensure the generated XML was compliant with the information interface.
 3. **Product software implementation:** Used to ensure that the XML interpreted within the product software is consistent with the information interface.

- **Component class definition:** Finally, class definition and relationships were defined in the Universal Modelling Language (UML). This simple component and object model was found to provide a good foundation for both component and client development.

6.6.1.3 System Implementation

System implementation was from the bottom tier up. Firstly, the database was implemented using Microsoft Access initially for speed in assessing system functionality. It will move onto a more powerful RDBMS as the system evolves. Because the business components all use ADO and ODBC to interface with the database, the change of engine should be straightforward.

All components in the business tier were developed using Visual Basic – this was considered the most productive environment with which to work. As there were no obvious performance bottlenecks in the business tier, little would have been gained from using C++, for example, rather than VB.

The user tier was also implemented in Visual Basic, for the similar reasons. It essentially provides thin clients for the system. There is little processing functionality within the clients, so the primary development task was GUI implementation. Visual Basic provided the most productive environment for this type of work.

6.6.2 Development Review

Overall, the Netscient case study provides a far more positive outcome than DOLMEN: development schedules were met, and functionally complete software was delivered. The primary goal of the first year of development for Netscient was to release version one of product software along with in-house processes to manage equipment profiling, and that goal has been met. Given

that similar development technologies were used in both cases it is interesting to assess where there were differences in approach which led to such very different outcomes.

In following review we look at things from a development perspective, and then from a project view. These different pictures help in assessing the impact of component technologies.

6.6.2.1 Development issues

Important aspects of the development process included:

- *domain- rather than product-orientated approach*

The domain centric approach certainly meant that initial development was a lot longer than would have been the case with a different approach. A lot of time was spent modelling the Netscient domain and encapsulating it in the form of an object framework. However, this time was recouped in application development, which was very productive once the framework was in place.

- *use of the different technologies to realise the various system elements.*

Three technologies were used within Netscient: an object framework for domain encapsulation, a component library to implement in-house personality management, and XML for inter-system communication. Clients for the in-house system and product software were developed as standard applications, incorporating either the object framework or in-house components. Third party components were also used in product software to provide additional functionality. As the two main elements (the object framework and personality components) had no dependencies, their distinct natures had no adverse effect upon development. The important issues which emerged from this mixing of technologies – the use of hybrid architectures and the differentiation between information and functional interfaces – are discussed in more detail in sections 6.6.3.3 and 6.6.3.2;

- *choice of an object framework for core domain analysis*

The encapsulation of core domain functionality into an object framework is judged successful at the present time. The framework has been successfully incorporated in two software products, and is currently being used in several others. However, an issue that may arise as the complexity of the framework grows is the amount of redundant code that is being included in applications. At present, the entire source has to be compiled into each application due to the monolithic nature of the implementation. One potential solution could be to break the framework into a number of sub-frameworks. Alternatively, component wrappers might be provided for different aspects of functionality. Objects are coded in C++, either approach would be possible without much modification to the source;

- *choice of component techniques for third party reuse*

This approach to incorporating non-domain specific functionality into Netscient product software has proved very successful. Firstly, it enables in-house developers to focus on domain specific functionality, but additionally it demonstrates how components can be used within a development process without dominating it. Using components, the developer avoids one of the problems with reusing objects – having to learn the object interfaces before reuse is possible. For components, while there is still a requirement to familiarise oneself with the actual interface definition, binding to the object, making object calls, etc. is all carried out in a standard, component-oriented way. To conclude, in Netscient, the use of components to encapsulate whole aspects of additional functionality has greatly enhanced development productivity;

- *choice of component techniques for in-house system*

This has also been fruitful. The components are beginning to be reused in administration applications. However, the choice of component techniques in this area was also used as a technology assessment. As mentioned in section 6.4, it was felt that the immediate use of components through product software development would be unwise, as component technologies were relatively immature. By firstly using components in-house, the capabilities of a component approach could be assessed and project personnel could gain skills in component development. The experiment has proved positive and as a result of this assessment, component technologies are going to be increasingly used in subsequent software releases;

- *appropriate design techniques*

As indicated above, interface definition, UML models, and XML specification were all used to good effect.

6.6.2.2 The Use of a Domain Oriented Approach

As discussed and demonstrated in section 6.4.1, the domain-oriented approach does not consider the products that the organisation wish to develop, but the domain in which they, and their products, will exist. By modelling the processes and entities with their domain, Netscient have provided themselves with the means to develop numerous, domain centred products from the same functional core. Currently, domain encapsulation is demonstrated in two packages. Firstly, an object framework, which encapsulates all of the entities that comprise a typical network management system and the functionality therein. This framework is currently in use in two products. It is also being extended and incorporated into future releases and also new products. The company believe that it provides a solid foundation on which to base new applications.

The second domain package encapsulates the in-house processing necessary to support Netscient's application suite through the management of network profiling and equipment personalities. Two clients currently exist, exploiting different aspects of the component library. The potential also exists to either use the component library in its current form for new clients (for example, remote Internet-based administration), or to extend the library to incorporate new functionality. Again, domain encapsulation has provided the foundation on which to build new applications without the need to alter the infrastructure.

When comparing this to the DOLMEN encapsulation approach (the conclusions from which are discussed in section 5.6), we have to conclude that the Netscient approach offers far more reuse potential than DOLMEN. While it is certainly true that other factors inhibit reuse of DOLMEN software – in particular, too many dependencies between components drastically reduce their utility. Nonetheless, the conclusion from comparison of the approaches used in the case studies is that domain encapsulation will generally lead to more reusable components that product encapsulation.

6.6.3 Issues Arising from the Use of Component Technologies

As mentioned in section 6.3, it was anticipated that other aspects of the use of component technologies would come to light as the project unfolded. The following are considered the most important of these.

6.6.3.1 Components in an SME

Netscient demonstrates the effective use of component standards to facilitate third party reuse - the project demonstrates this through the reuse of extra-domain functionality developed by other software vendors. This has enabled a vertical domain software house to focus their own development on domain functionality buying in supporting functionality from third party sources.

Of course, third party reuse is possible with other technologies (for example, object frameworks) – but it is particularly convenient with components.

The ability of domain specialists to focus on their own area, buying in supporting functionality is critical for an SME. SME software houses can focus their development effort upon domain specific knowledge and, through developing using component standards, expose their domain knowledge to other SMEs. A scenario could be envisaged where a number of SMEs, each with specialised domain knowledge, might share their experience via component techniques to achieve far more than possible by a single company. Such *virtual corporations* might hope to compete on equal terms with the large software houses that have far greater resources at their disposal.

6.6.3.2 Information interfaces

Perhaps one of the most important findings in the Netscient case comes from the separation of functionality and information when considering distribution. Component technologies can certainly provide the functionality to allow the passing of complex structured information across distributed systems. Indeed in the DOLMEN case, all information was passed as parameters in component calls. Even stream communication – communicating information over a session connection – was dealt with using a CORBA server. However, this resulted in a problem – the complexity of interface definitions was greatly increased to accommodate the information being passed between components. As the information was passed using CORBA object calls, the information had to be passed as parameters within a function call. At best the communication would require a single structured data type, at worst numerous structures, all with structures nested within them. An example of this is given in Figure 6-9, where a structure is embedded in a sequence, which is then embedded in another structure. The use of such a construct within a C++ implementation can easily lead to problems. Memory management within a component system is already complex due to the distributed nature. When complex types are involved, bugs are all too

easily introduced. A single error within a structure can cause crashes that are very difficult to diagnose.

```
struct FlowDescriptor {
    FlowId flowid;
    StatusSB flowstatus;
    SFEPIId flow;
};

typedef sequence <FlowDescriptor>
FlowList;

// stream binding (SB) description

struct SBDescriptor {
    SBId id;
    FlowList flows;
    StatusSB sbstatus;
    t_UserId party_a;
    t_UserId party_b;
};
```

Figure 6-9 - Typical complex information structures in DOLMEN

This lesson was heeded in the Netscient case, and information passed between components was restricted to the minimum necessary. A policy of separation of *functional interfaces* and *information interfaces* was developed. The two kinds of interface are best engineered using different development technologies. A functional interface – enabling access to functionality provided by a component – is best implemented using component technologies. This is one of the strengths of component approaches.

An information interface – enabling clients and components to exchange structured data - is less well served by component technologies. Problems occur when the complexity and volume of information reach such a level that the communication overhead gets too high. In the Netscient

project the use of XML was found to be more appropriate. XML was devised to exchange structured information in the context of the Internet. However, it is equally applicable to structured data in any distributed system. As TCP/IP continues to establish itself as the de-facto networking standard, the use of Internet technologies can usefully complement component technologies as both use the same network standard. Moreover, the dominant XML parsers (IE4 and IE5 parsers) are themselves implemented as COM object models, and are therefore easily employed in a component based environment.

Thus, experience in the Netscient project suggests that whereas CORBA and DCOM are sold on their support for the development of distributed systems, the use of those facilities may lead to unnecessarily complex and inefficient implementation. Some aspects of distributed development are well served by a component approach, but others are better handled by different technologies.

6.6.3.3 Hybrid Platforms and Mixing Development Technologies

The Netscient project also featured a more general mixing of development technologies to form a hybrid platform (see section 6.5). As mentioned earlier in this chapter, it is usually considered that the adoption of component technologies into a development process has to wholly embrace component technologies in order that the use of such techniques is successful. The Netscient case, which achieved all of its first year goals while mixing development technologies, shows that this need not be the case. The Netscient platform mixes Internet, object and component technologies effectively, achieving a great deal of software reuse through the exploitation of these techniques. While there are obvious system boundaries between the primary object and component implementations (i.e. one was for product software and one was for in-house software), all three technologies are employed successfully in product software.

In DOLMEN, there was almost an insistence by management that everything had to be component-based, even when it was apparent that some parts of the DOLMEN architecture would only reside in one place and be used in one context. In these cases (for example, the Stream Interface [160] was simply a function library performing functions similar to a TCP/IP stack (i.e. send, receive, etc.)) it may well have been better to implement as standard objects within the components. As component implementation languages tend to be object-oriented (e.g. C++, Java) this would have been straightforward.

The most evident conclusion to draw from this finding is that it provides some argument against the commonly held industry belief that component technologies have to be wholly embraced in order to be effective (for example, see [28]). However, the implications could be more widespread. As stated above, the majority of literature relating to component-orientation encourages a replacement of existing technologies with these new techniques. When considered in the context of the two case studies, the wholehearted component-oriented approach suffered far more problems than the hybrid approach. What is evident is that while component-orientation does provide some extremely useful techniques for software development (for example, standards for reuse, the distribution of functionality), it is not a panacea. Using it to its strengths, and using other techniques for other areas, provides a more effective development approach.

6.7 *Consideration of Findings Against Case Propositions*

6.7.1 Proposition 1

Adopting and using component technologies in software development processes will affect process activities

The case also positively identifies a number of issues that the use of component technologies introduces to development activities. Early on in the development process, requirements analysis

was greatly affected – the identification of functionality outside of the core domain resulted in the need to identify third party components that could be used, rather than considering implementation of functionality with which organisational developers have no expertise.

We also have evidence of the effect component technologies in both design and implementation activities. The use of interface definition and modelling within the Netscient case provided useful tools to the developers of separate, but interacting, software elements. The success of these techniques could possibly be attributed to stricter version control and greater developer communication in the event of design changed. Implementation activities, in particular distributed development, was aided a great deal by both the component standard and services, which enabled a swift and scalable implementation.

As with the first case study, consideration should be made to how much contribution was made by the choice of development approach to the issues that arose from using component technologies. The more positive results in the use of technologies within development activities are undoubtedly as a result, in part, to the more iterative nature of development adopted by Netscient, as it gave the opportunity for risk assessment and introduce contingencies before problems occurred.

Therefore, we should consider development from this proposition carefully – we can state that component technologies do have an effect upon development activities, but we must also state that the nature of the development process also contributed to the relative success of the use of components.

6.7.2 Proposition 2

An awareness of the issues involved in the adoption and use of component technologies can ease their integration

With this proposition, we have a far more positive outcome than that of the DOLMEN project, as a result of having a more informed decision making process when considering component technologies. Additionally, directors in Netscient were more cautious in their use of component technologies, trialing them initially on non-essential developments before considering their use with product development. The trialing also enabled developers to gain hands-on experience with the technologies before their use in product development. This is an important lesson to draw from the Netscient case – technology trialing can enable the development of experience away from essential software development within an organisation.

Therefore, this proposition can be confirmed, and should be developed to consider how this awareness can be promoted.

6.7.3 Proposition 3

A domain-oriented approach to component development provides a greater degree of reuse than a product-oriented view.

A domain oriented approach in Netscient has certainly promoted effective reuse with both in-house and product systems. In comparison to the product centric view of DOLMEN, the level of reuse at Netscient is far higher. In developing the proposition, however, we should consider the role component-orientation played in this successful approach to reuse. Domain orientation has been demonstrated effectively with other development technologies (a very obvious example would be the Microsoft Foundation Classes object framework for Windows development [83]).

Therefore we cannot consider component-orientation to be the driving force behind a successful domain oriented reuse strategy. However, the case study does demonstrate that the machinery required for software reuse can be achieved, due to the mechanism they provided, by component-orientation (such as binary level reuse and common interfacing via the component standard).

6.7.4 Proposition 4

Similar issues with component-orientation occur when using different technologies from the same field (i.e. Microsoft based, rather than OMG based technologies)

This proposition guides the testing of the theory that the problems of DOLMEN directly relate to component technologies. While the level of complexity within the DOLMEN project was high, this was related to the nature of management interfaces rather than the implementation of the software component themselves. Within Netscient the management interfaces were simpler (i.e. interfacing to databases rather than telecommunications hardware) but the level of implementation was similar. Therefore, the first impression through the comparison of experiences within each case study could be that CORBA technologies do provide more problems than COM technologies. However, we should consider whether this issue was directly related to the selection of technologies (i.e. a COM- rather than CORBA-based approach), or whether it reflects greater knowledge in the use of component-orientation and a more flexible development approach. If we hypothesise that the choice of component technologies does affect the experience of use component technologies, the context of each case study does not allow this to be tested.

Therefore, outcomes based upon this proposition promote further examination into the use of the different types of component technology. While the case study demonstrate a better experience using Microsoft technologies, there are too many variables within the case for this outcome to be considered generalisable at this stage. Additionally, this proposition does encourage consideration

of component technologies within other vertical domains would be useful in considering whether they reflect the nature of an industry or whether they have universal potential.

6.7.5 Proposition 5

Issues in the DOLMEN case study can be avoided through greater knowledge of the technologies involved

This proposition complements the second, related to a need for awareness of the issues involved in the use of component technologies. In a direct comparison between case studies, we can consider the greater knowledge that Netscient had before using components aided in the avoidance of the sorts of problems experienced in DOLMEN. Additionally, the all embracing use of components by DOLMEN was not used in Netscient, due, in part, to a greater knowledge of the strengths and weaknesses in the use of components.

Caution is needed in interpreting these outcomes from the proposition, since the nature and lack of control variables within a case study approach means that it is difficult to conclusively demonstrate fact from case study findings. However, it is worth saying that the issues that occurred in these cases *could* happen in the use of component technologies. The question is how best to communicate this, and how to separate the important, common issues from the more idiosyncratic events which resulted from extraneous factors.

6.8 Summary

The Netscient case was interesting as a vehicle for testing of some theories developed from the DOLMEN study, but also in its own right. It reinforced some findings regarding the use of component technologies, which will enable an more effective focus of issues when considering the communication of experience (see chapter 8). It also confirmed that using components for a

domain-orientated approach to software reuse is likely to offer greater reuse potential than a product-oriented approach used in DOLMEN.

Third party reuse has been demonstrated as a very powerful technique to exploit the knowledge and developer resource of other organisations. By choosing a component-oriented approach to the reuse, the time, and developer effort, required to integrate the third party functionality into organisational system can be greatly reduced, when compared to object libraries, due to the standard nature of component-oriented reuse. As the third party components were implemented to the same standard as that being used by Netscient developers, they could be confident that standard interfacing techniques would enable swift integration.

However, the most interesting results have come in unexpected areas, where perceived wisdom regarding the use of component technologies was ignored. The separation of functionality and information is one example. Using different technologies, the strengths of each could be exploited. More generally, Netscient showed that the mixing of component technologies with other techniques need not impair their use in any way and may be decidedly beneficial.

Consideration of these outcomes against case propositions has provided validation of some of the outcomes of DOLMEN, resulting in some theories being developed and some being rejected as peculiar to the DOLMEN project. Additionally, further theories in the adoption and use of component technologies have been arisen from this case study.

In order to direct the thrust of communication of issues regarding component technologies further, there is a need to draw from the experience of others. While the case studies provide depth of analysis, they lack strong external validity. Therefore, a practitioner survey was carried

out further determine the generalisability of issues from the case studies. The following chapter describes this survey and its results.

In the last of the chapters related directly to data collection within the research programme, the need for, and the development and execution of, a practitioner survey is discussed. Literature review has highlighted the problems with generalisation from case study findings and this is addressed through the surveying of others with experience in the use of component technologies. The survey development was guided by theories developed from case study findings, thereby allowing for a validation of certain theories and the focusing of result development upon common issues.

7. Practitioner Survey

As a final strand of investigation within the research project, a practitioner survey was carried out based upon case study propositions. Comprehensive generalisability could not be determined from two separate case studies, and it was important to determine the frequency of problems in the adoption and use of component-orientation technologies. Anecdotal evidence from industry peers and some emerging literature both suggested that the case studies were certainly not exceptional within the field. For example, Herbsleb and Grinter [67] detailed a case study within Lucent Technologies where comparable experiences occurred in a project similar to DOLMEN. While the focus of their case study was the need for communication within distributed development teams, we could identify a number of issues regarding integration, assumption and lack of technology trialing that go some way to confirming at least some commonality of experience in the use of component-orientation.

The survey was, therefore, conducted in order to obtain quantifiable opinion on case study results and to assess the normality of experiences within the studies. By focussing upon practitioners the results assessment could be very much realistic to the development industry. The objectives of the survey were as follows:

1. To assess practitioner experience in the learning, adoption and use of component technologies

2. To compare practitioner experience with case study propositions and findings
3. To aid in determining the generalisability of case study findings
4. To identify areas of weakness in the adoption and use of component technologies

7.1 Survey Approach

It was decided that rather than use a traditional survey approach (for example, postal or telephone), an online, World-Wide Web (WWW) based survey would be used. The survey was held on a WWW server and presented to potential respondents as an online form that they could carry out via a browser and submitting the results electronically. An online approach was considered beneficial for a number of reasons:

- **A format that appealed to the target audience** – it was considered a suitable format for a survey as the target audience would be technical and IT focussed.
- **Storage of results** – Results from each survey were stored in a text file on the server that could be easily imported into data analysis software once the survey was complete
- **Reducing time to contact and respond** – in comparison to a postal survey, the time taken to send out the survey and obtain responses could be reduced using an online method.

The initial survey was piloted in order to refine its construction and improve its readability. Researchers with component experience from the Network Research Group, University of Plymouth were used for this pilot. Potential respondents were contacted via email with a message briefly explaining the aims of the survey and including the URL of the survey. This enabled the recipient to go straight from reading the mail message to carrying out the survey. With the combined email/online approach, the respondent can carry out the survey without leaving their PC as soon as they receive the email.

It was important to obtain responses from practitioners actively involved in the development of component-based systems. As potential respondents were also to be contacted via email, a list of email addresses was required. The most effective information resource in addressing both of these requirements in obtaining responses was to go to mailing list archives in the area. Mailing lists are used in developer communities to share ideas and ask questions related to the list topic. List providers tend to hold archives of previous questions and answers for reference, generally organised in either month or year sections. Therefore, by going to list archives, email addresses could be obtained from developers who were active and experienced in the area of component-based development. In general, questions and discussion from the chosen archives (CORBA-DEV and DCOM@discuss.microsoft.com) asked in the mailing lists were also complex in nature – therefore demonstrating a good level of knowledge in the area. Additionally, two personnel from each of the case studies completed the survey to see whether responses from project developers would reflect case outcomes.

7.2 Survey Construction

The survey is included in appendix D. The mix of questions was intended to explore issues arising in the case studies, without guiding the respondent in their answers. It was divided into the following sections:

- **About you:** General information about the respondent (name, job title and organisation). It was stressed that this information was optional – while the information from the survey was not particularly sensitive, some of the target audience may wish to be anonymous.
- **Regarding your use of component technologies:** To establish the respondent's experience using component based techniques. This was done for a number of reasons:
 - To establish the degree of experience in using component based techniques in practice

-
- To determine the types of component based techniques used (in order to establish an differences in CORBA and COM based experience – reflecting case study two’s first proposition).
 - To determine on what types of projects component based techniques were used, and in what vertical domains (to determine the spread of use)
 - **Regarding your learning of component technologies:** In order to determine how the respondent learned about component techniques to gauge the approaches used and how best to integrate results of this research project with those approaches. Also, to bring to light any particular problems with learning about component techniques.
 - **Regarding component technologies and the software development process:** Focusing more upon findings from the case studies – it is important to determine whether case study findings reflected the norm or phenomena within component-based projects. Initial questions determine the respondents own experiences integrating and using component techniques within their own development processes, while the final set of questions all relate to specific aspects or activities within the development process.

7.2.1 Question Construction

In general, the survey consisted of closed questions⁹. This meant that analysis could be carried out swiftly as answers could be grouped by response. It was only when further elaboration was required based on a closed response that some open questions were used. In these cases, analysis of responses attempted to group answers into specific classification for result presentation. For initial sections of the questionnaire, most responses required only a yes/no response. While a

⁹ A closed question will present a set of responses (e.g. “Yes/No”, a list of responses), rather than an open question where the answer is left entirely up to the respondent [138].

richer response could have been obtained using bipolar questions for the entire survey, it was decided only to use these in the final section for two reasons. Firstly, the start of the survey aimed to establish key concepts, a level of agreement was not required until questioning related directly to case study theories. Secondly, there was a conscious attempt to make the questionnaire as straightforward to complete as possible. Presentation of a large number of complex questions could make it appear more imposing and therefore adversely affect the response rate.

However, the final section of the survey ("Regarding component technologies and the software development process") did take the form of bipolar agree/disagree questions, where a statement is presented and the respondent is asked to what degree they agreed or disagreed with the statement. Traditionally, these questions can suffer due to acquiescence [138], where a respondent tends to agree with the statement. It was important with this set of questions to get opinion based upon practitioner experience, not simple agreement, so the problem of acquiescence was addressed in two ways. Firstly, rather than simple agree/disagree responses, they were divided into a range of responses (from "strongly agree" through "no opinion" to "strongly disagree"). Secondly, the statements were not always stated as positive, and were not always stated to reflect case study findings (for example "project management is unaffected by component technologies" and "component development makes system deployment easier"). Based upon survey responses, it would seem that these attempts to avoid "guiding" the respondent to reflect case study findings were successful.

7.3 Survey Response

Two hundred practitioners were emailed during March 2000. Forty-three responses were obtained, providing a response rate of 22%. It was also interesting to note comments received from respondents regarding the survey. Respondents were given the opportunity to include their email address in the submitted survey if they were interested in the survey results. Forty-two out

of the forty-three respondents stated an interest in results. Emails received from some respondents also reiterated their interest in the results and their interest in the research programme in general. Additionally, a few people who felt that they did not have the technical experience to complete the survey also expressed an interest in the results. A list of respondents is included in appendix E.

Results analysis is presented on two levels – firstly, basic responses to individual questions are considered. However, with some questions, basic analysis is extended to include trends based upon other responses, cross question analysis and consideration against case study propositions and findings.

7.4 Survey Analysis

7.4.1 Regarding your use of component technologies – establishing respondent type

1. How long have you been using component-oriented techniques?

Statistic	Value(years)
Min	0
Max	13
Mean	3.93
Std. Dev.	2.91

Table 7-1 - Statistics regarding experience with component technologies

Table 7-1 provides the basic statistics for experience in the use of component technologies. As expected with an emerging technology, the mean value is quite low. Additionally, a few responses of 8 or more years distorts the distribution – the entire distribution has a skew value of 1.28. Table 7-2, below, provides statistical information for the distribution with high values

(8+ years) removed. Overall, 5 values were removed from the distribution. The skew value is greatly reduced (0.24), providing a far more realistic mean for the majority response.

Statistic	Value(years)
Min	0
Max	7
Mean	3.11
Std. Dev.	1.72

Table 7-2 - Statistics regarding experience with component technologies (high values removed)

2. How long have you been developing software in general?

Statistic	Value(years)
Min	2
Max	34
Mean	11.1
Std. Dev.	6.82

Table 7-3 - Statistics regarding general development experience

There is a good spread of experience among respondents, ranging from relative newcomers to extremely experienced developers. Once again, a few very high values distort the distribution, resulting in a skew value of 1.72. Their removal (4 values of 20+ years), detailed in Table 7-4, results in a far less skewed (0.18) distribution providing more meaningful majority response statistics.

Statistic	Value(years)
Min	2
Max	18
Mean	9.5
Std. Dev.	4.1

Table 7-4 - Statistics regarding general development experience (high values removed)

3. What component standards have you used?

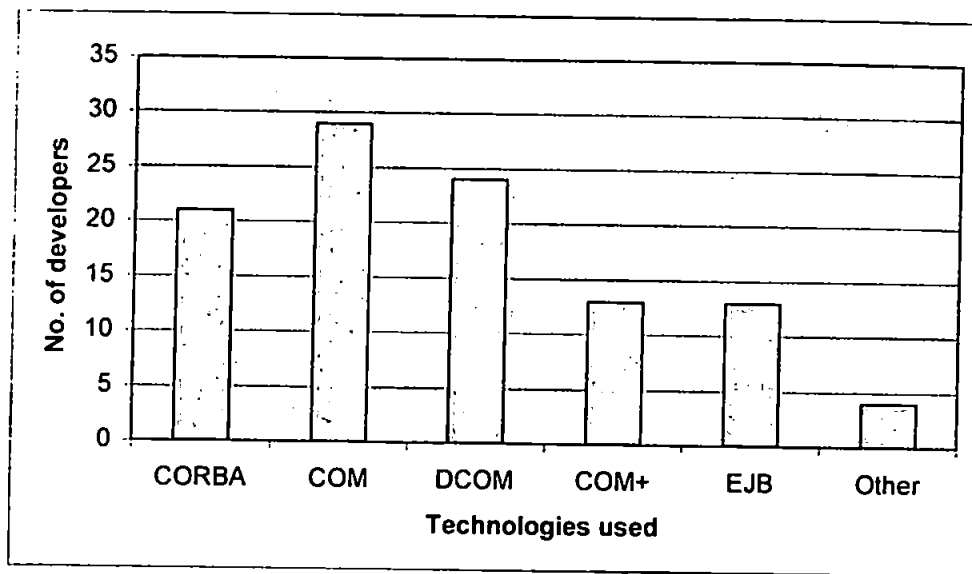


Figure 7-1 - Component technologies used

A fairly predictable result, with COM, CORBA and DCOM being dominant. The number of respondents with COM+ experience is somewhat surprising, particularly as its availability at the time of the survey was still quite limited. The inclusion of the DCOM mailing list, which generally addresses highly complex aspects of component based development, involved practitioners very much on the leading edge of the field.

Responses for "other" included two other Java technologies – Remote Method Invocation and basic JavaBeans, and also XPCOM (an open source COM implementation provided by the Mozilla organisation – see <http://www.mozilla.org/projects/xpcom/>). Another was a set of library components based upon, but distinct to the CORBA component model. A final "other" response – EntireX – could be regarded as a DCOM response, as it provides an implementation of the standard on UNIX platforms.

Another aspect of response, important when considering trends and cross-question analysis, is the distinction between CORBA and COM developers. An outcome from the case studies was that there may be differences in experience depending on whether CORBA- or COM-related technologies are used. For this analysis, COM, DCOM and COM+ were considered COM related technologies and CORBA & EJB were considered CORBA related technologies. Figure 7-2 illustrates the experience of respondents. The “neither” response came from the respondent who had used the “CORBA-like” model.

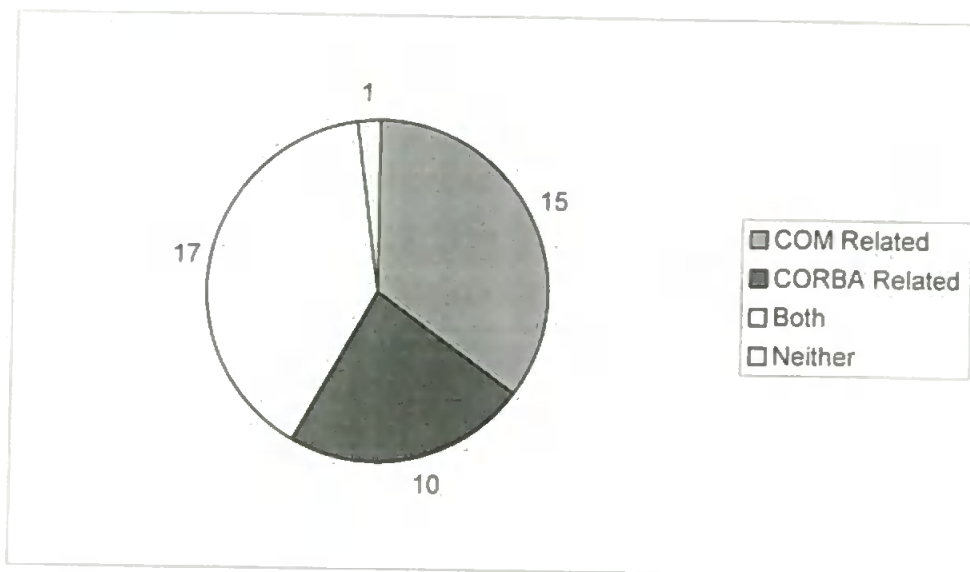


Figure 7-2 - COM & CORBA related experience among respondents

One significant outcome from this grouping emerged when considering the experience classification of the respondents by type (see section 7.4.1.1). In the case of “COM” and “Both” respondents, there was a spread over all three-experience classifications. For CORBA respondents, there were no respondents who were “very experienced” and 50% were intermediate. Also, it was interesting to note that “COM” respondents had the highest proportion of “very experienced”.

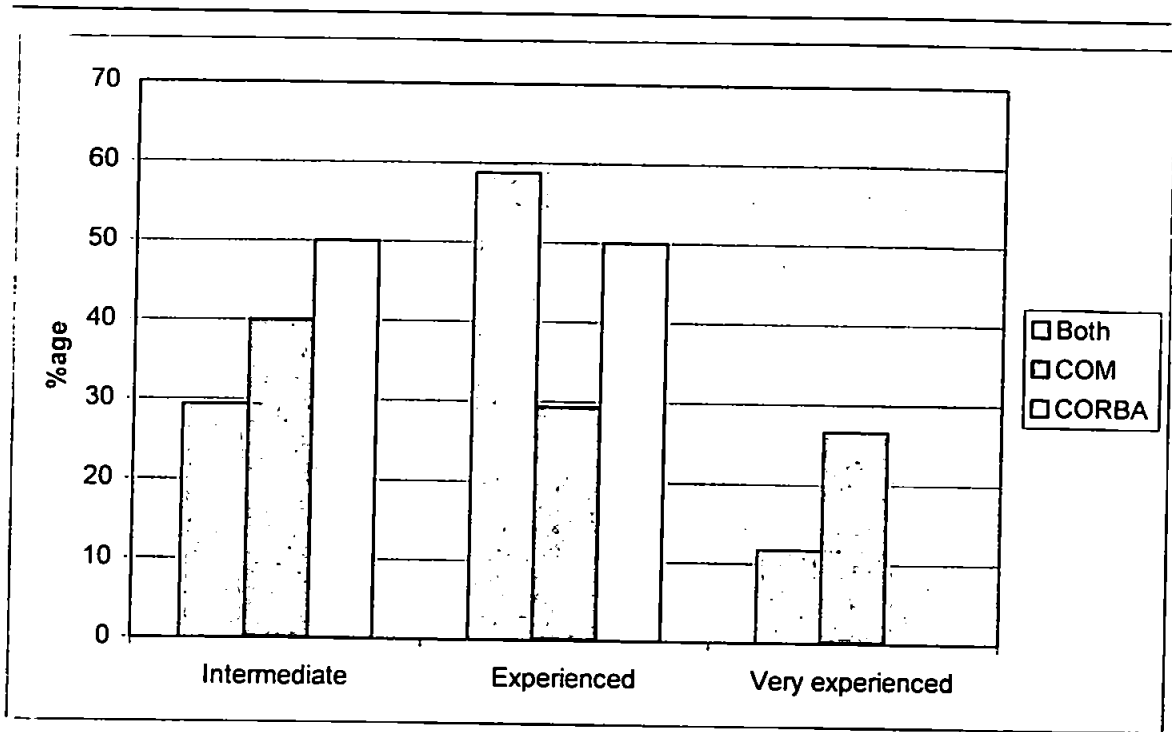


Figure 7-3 - Experience classification of respondents

4. What component tools and technologies have you used?

There was a wide range of responses to this question. While the majority of COM related technologies centred on Microsoft’s Visual Studio, there was great variety with CORBA. Visibroker and Iona being the most popular tool vendors, with approximately 40% of CORBA experienced respondents having used them.

5. On how many projects have you used component-oriented techniques?

Statistic	Value(number of projects)
Min	1
Max	30
Mean	8.42
Std. Dev	9.26

Table 7-5 - Statistics regarding number of projects where component technologies have been used

There is again a good variety of responses, and considerable experience of component technologies among respondents. Here again, there were two high values that skewed the distribution (1.72), and these were removed to get more realistic statistical values. These are provided in Table 7-6:

Statistic	Value(number of projects)
Min	1
Max	15
Mean	6.3478
Std. Dev	3.9267

Table 7-6 - Statistics regarding number of projects where component technologies have been used (high values removed)

6. On what scale of project have you used component-oriented techniques?

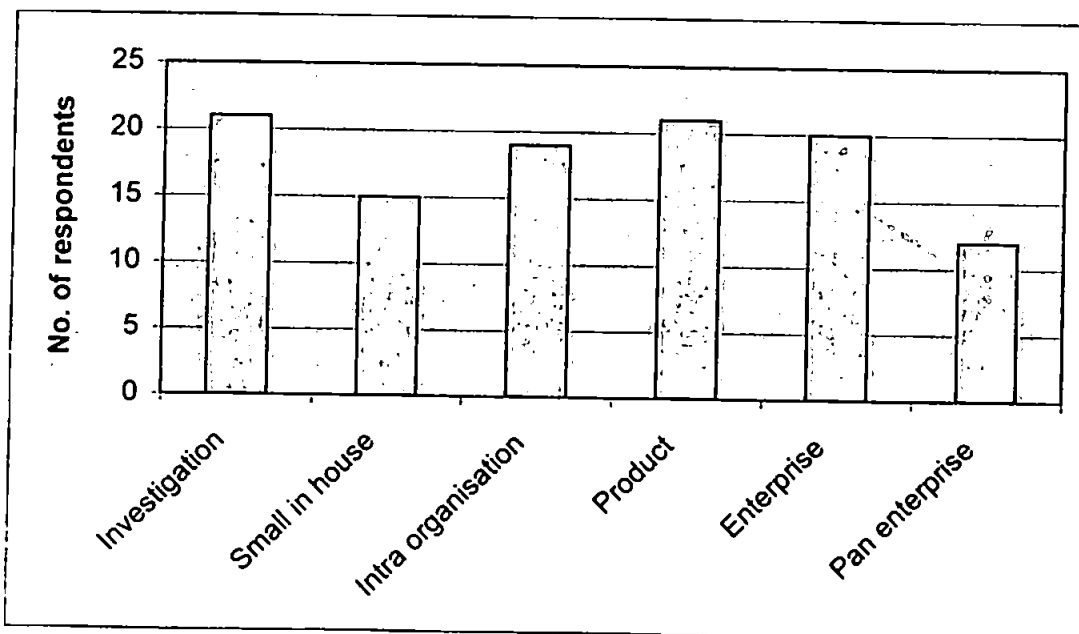


Figure 7-4 - Use of components in different project types

Figure 7-4 details the spread of types of project that have used component-based techniques. The high value in investigation would suggest that many projects that assessed technologies before using them on a larger scale. Only a single respondent has used component technologies solely on

investigative projects. A surprising, but encouraging, number of respondents had used components on large-scale (product, enterprise or pan-enterprise) projects. This is very useful as it demonstrates real world use of component technologies.

7. In what vertical domains did these projects reside?

Figure 7-5 illustrates responses to this question. Unsurprisingly, IT services and telecommunications are the dominant industries in which components are being used. Additionally, the high level of responses in the financial sector reflects the high level of resource available in that sector for investment in new technologies. "Other" sectors described in the responses include experimental/research, open source operating systems, CAD and pharmaceutical. The varied response across many different vertical sectors would suggest the inherent generic applicability of components. As both case studies focused upon the networking/communications domain, to obtain responses from other domains is useful in further examining the generalisability of case study findings.

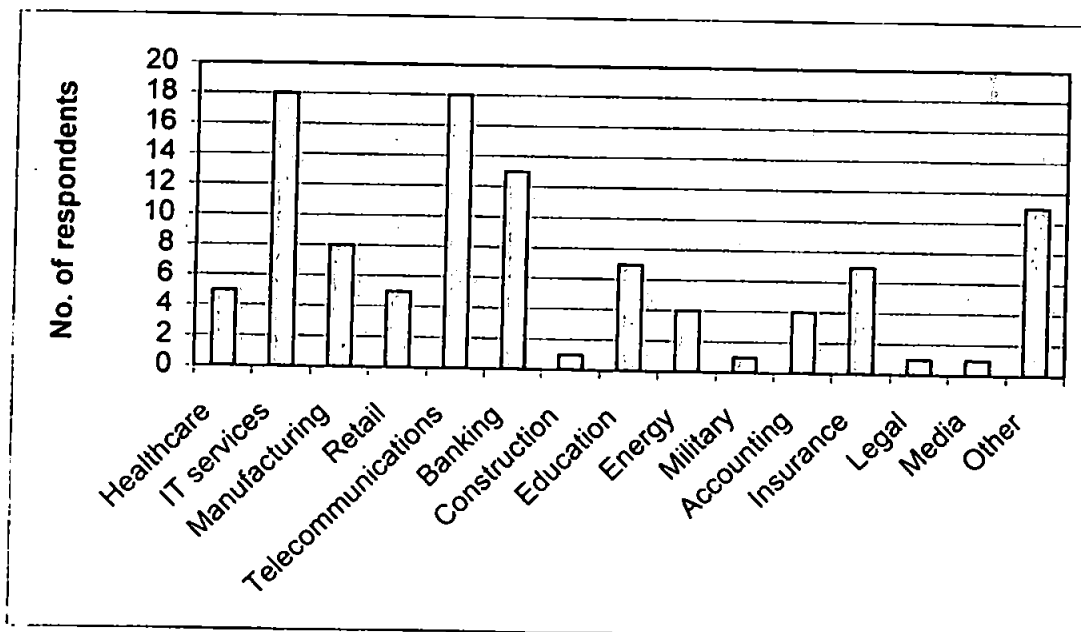


Figure 7-5 - Use of component in vertical sectors

7.4.1.1 Model Respondent and experience classification

Based upon average and majority responses, a model respondent can be defined (note that means are based upon modified values with high values removed to reduce skew). The model respondent is detailed in Table 7-7, and provides a benchmark with which to compare individual responses – this is developed when considering responses from Netscient and DOLMEN project workers in section 7.5.1.

Experience with component technologies	3.1 years
General development experience	9.5 years
Component standards used	CORBA, COM & DCOM
No. of projects using component technologies	6
Types of projects	Investigation/assessment & product
Vertical domains	IT services, telecommunications & banking

Table 7-7 - Model respondent

The definition of an experience classification for each respondent enabled the development of another comparative measure. Based upon answers from the first section of the questionnaire, respondents had an “experience rating” assigned. This provided a value to use in cross-question analysis with the other sections of the questionnaire. The experience rating defined a respondent as intermediate, experienced or very experienced. Figure 7-6 illustrates the process in determining an experience rating. As stated in the figure, the weightings assigned to each parameter reflected the perceived relative important in determining the level of experience a respondent had specifically with component technologies. Component experience is obviously the most important value, and has the strongest weighting assigned. Project variety and number of projects

both share an equal weighting and are included as they demonstrate a breadth of experience in the use of components. Development experience is also included as a parameter, although not strongly weighted, as experience with other development techniques may help in the learning and use of component technologies (in particular experience of object-orientation). Note that the intention of the experience rating is to give a simple quantifiable value for use in the evaluation of future responses – it is not intended to be a precise measure.

Determining an "Experience Rating"

Determine a "Project Variety" value

A value based upon the types of project on which the respondent has worked, with weighted values assigned to each type of project – these values reflected the complexity of each project type.

Investigation = 1

Small in-house = 2

Intra organisation = 3

Product = 4

Enterprise = 4

Pan-enterprise = 5

Project variety = sum of the above

Determining an "Experience Value"

Based upon weighted summation of component experience, development experience, project variety and the number of projects worked on. Weightings were based upon the perceived importance of each value in determining experience with component technologies.

*Experience_value = (component experience * 0.5) + (development experience * 0.2) + (project variety * 0.4) + (no. projects * 0.4)*

Determine "Experience Rating"

<i>Intermediate</i>	<i>0 <= experience value <= 6</i>
<i>Experienced</i>	<i>6 < experience value <= 15</i>
<i>Very experienced</i>	<i>experience value < 15</i>

Figure 7-6 – Determining an experience rating

Figure 7-7 provides the distribution of intermediate, experienced and very experienced respondents.

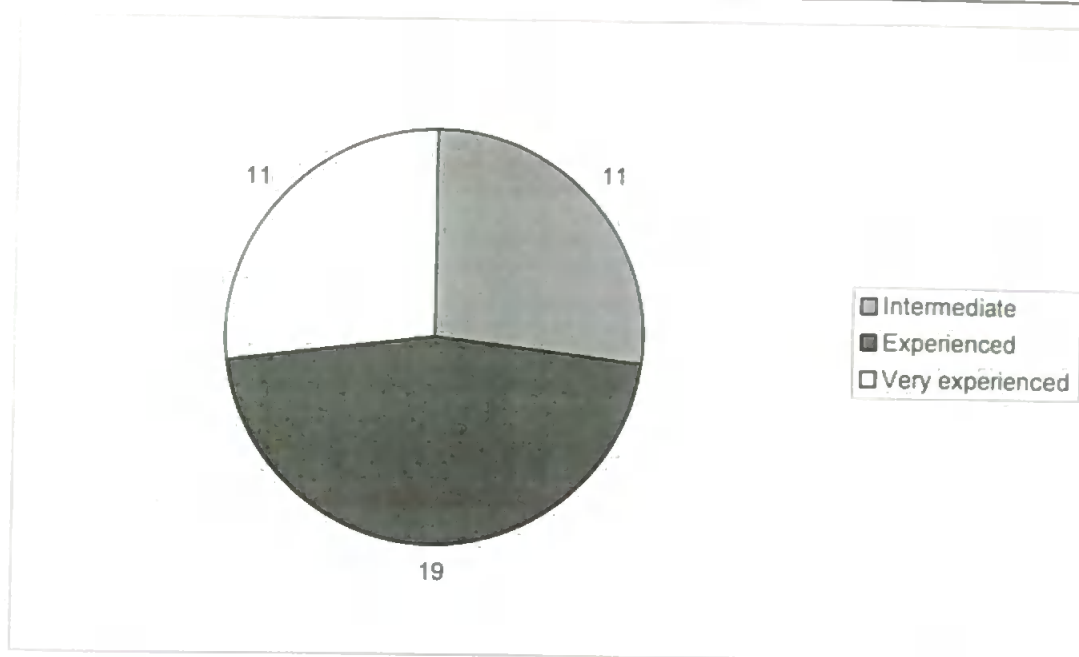


Figure 7-7 - Distribution of experience ratings among respondents

7.4.2 Regarding your learning of component technologies – establishing learning approaches and common problems

8. How did you learn about component technologies?

Figure 7-8 illustrates this result and demonstrates an expected outcome. Mainstream development still considers component technologies very new, and reflects this in the lack of training available in the area. Therefore, practitioners wishing to learn about such techniques have to use literature and practical projects. “Other” responses were discovery/invention (from a respondent who has been involved in component-based development for a long time), in-house mentoring, research and self-study.

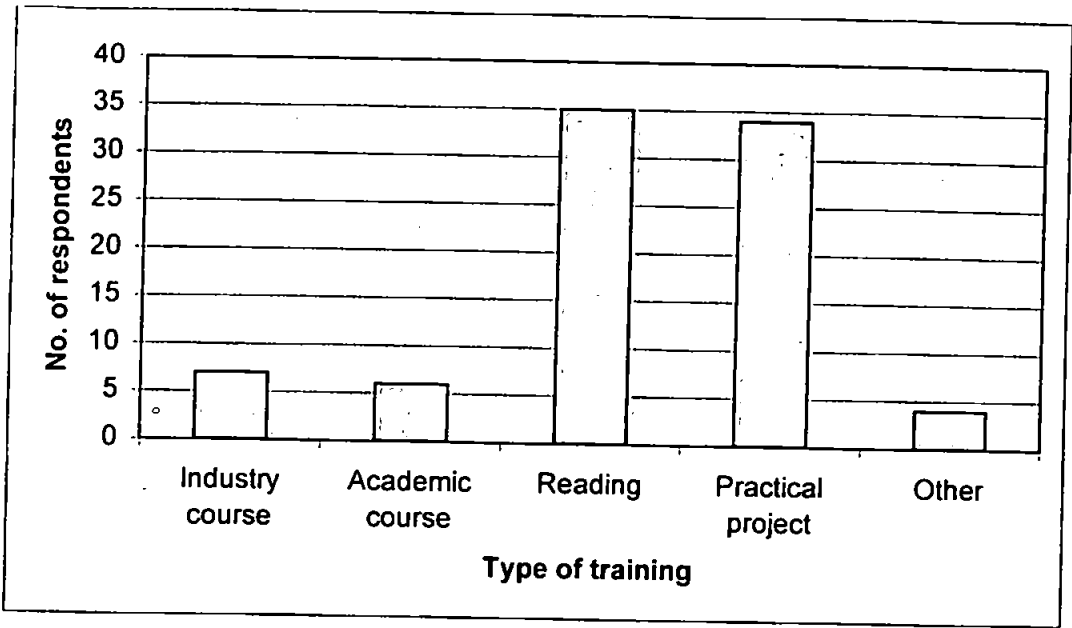


Figure 7-8 - Learning about component technologies

9. Did you experience problems when learning about component technologies?

This result is significant in considering the development of component-orientation into a mainstream technology. As a contributing factor in the adoption of a technology, both diffusion of innovations [133] and organisational learning [139] theories comment upon the complexity of a technology being a barrier to adoption. If the perception of component technologies is that they are difficult to learn and therefore complex, their adoption will be significantly hampered. In terms of the results of this survey, almost seventy percent of respondents experienced some difficulties in learning about component-orientation.

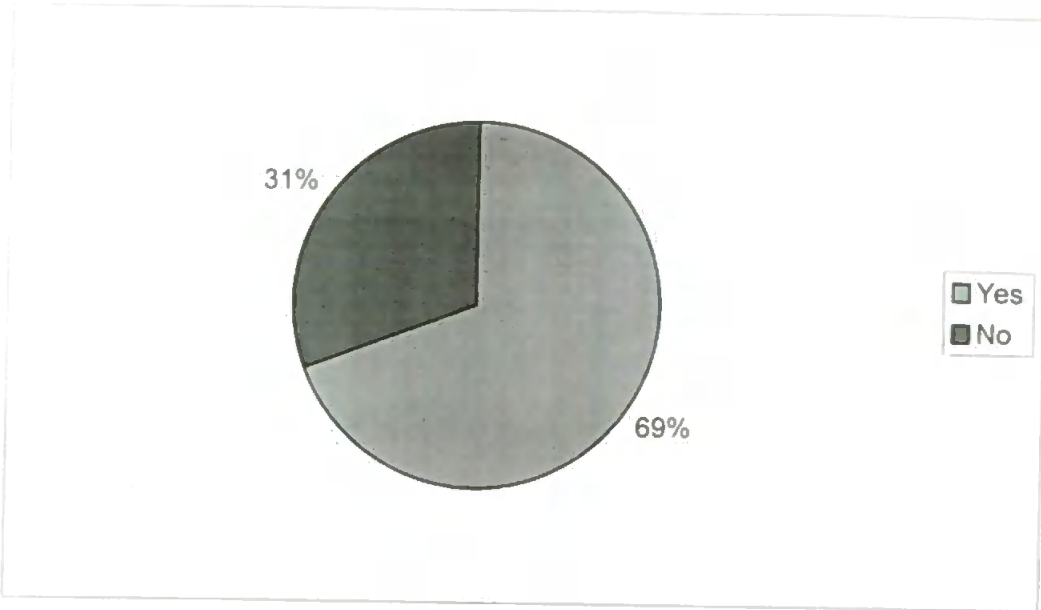


Figure 7-9 - Problems when learning about component technologies

10. If, yes, were these problems related to (concepts, technologies, differences between the two, other)

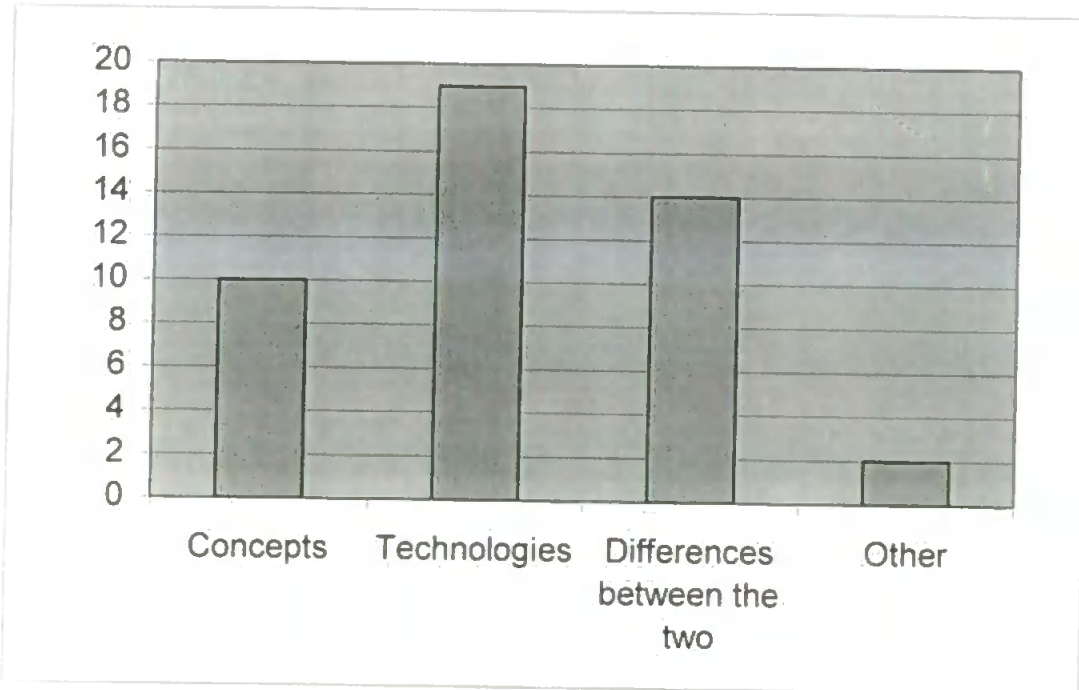


Figure 7-10 - Problems when learning component technologies

Developing from the identification of problems with the learning of component technologies, the main problems seem to relate to the technologies themselves and also the reconciliation between concepts and technologies. The problems of this reconciliation have been emphasised in both the case studies (for example, [129] is a deliverable from the DOLMEN project related to the development of the CORBA platform) and also industry in general [106, 132]. The response from the questionnaire further highlights this issue. The majority of elaboration provided by respondents also focuses around this area, with comments such as:

"Concepts are not well-understood in practice and thus, are not well supported."

"The tools and technologies, especially early on, were not mature enough to support the concepts."

"Incompatible vocabulary among various technologies; introduction of unnecessary vocabulary..."

It seems that while the concepts regarding component-orientation are reasonably clear, they have proved less easy to put into practice.

"Other" responses focussed upon problems with documentation, in particular for COM technologies. Comments included:

"reasonably steep learning curve for COM. The documentation seems obfuscated."

"Most difficulty with product documentation of their Component Model."

"COM is poorly documented - the whole thing's a mess!"

These comments are interesting, especially in consideration of the responses to the following question, which was about the literature available.

11. Did you find the literature about component technologies useful when learning about it?

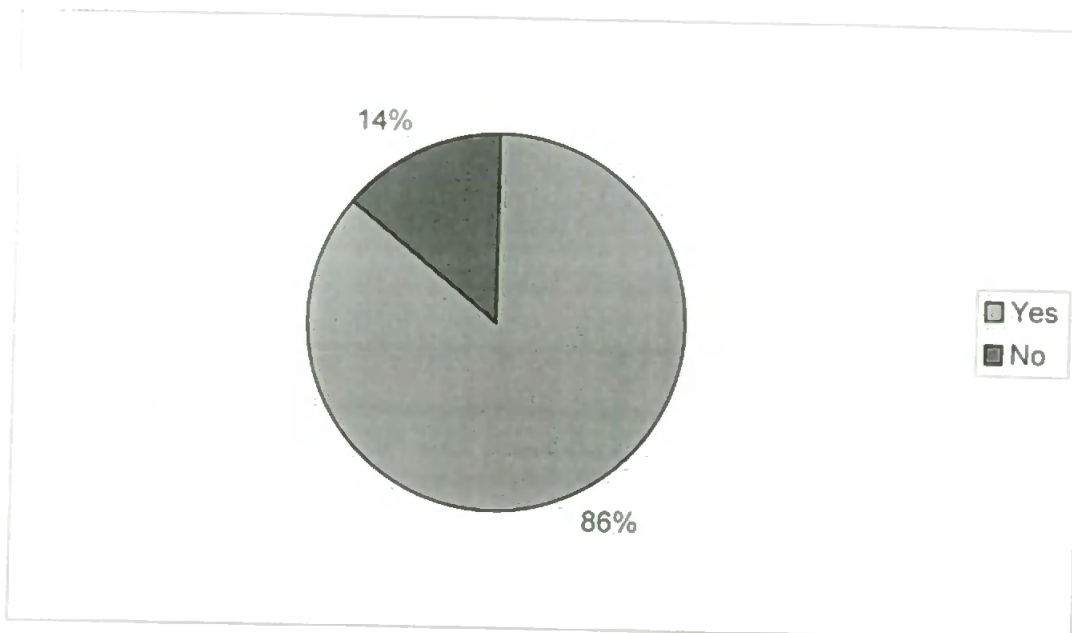


Figure 7-11 - Was the literature useful when learning

There seems to be some discrepancy between this and the previous question – while a lot of respondents had difficulties learning about component-orientation, in particular with respect to technologies and the differences between concepts and technologies, the vast majority of respondents did find literature related to their learning useful. There is little change in distribution when considering only those respondents who used “reading” as an element of their learning process. This would suggest that problems were not related to the documentation itself, but perhaps differences between what the literature said and what the technology would do.

However, comments from those who did not find the literature useful also reflect the problems between concepts and technologies:

“Too vendor specific, not grounded in reality”

"Inconsistent. Some techniques were shown in MFC, others in ATL. Sometimes took days to discover the correct interface for the job. CORBA documentation virtually non-existent!"

"Early on, nothing was available. Much literature is still too vague to adequately explain the concepts and get programmers using them effectively. I've seen a lot of messes result from this training issue."

"There is very little simple, practical documentation. Either it is highly technical or relatively simplistic with no practical applications"

"Too fragmented. Not easy to get my hands on a single source."

12. Would it have been useful to be able to draw from the experience of others that had used component technologies?

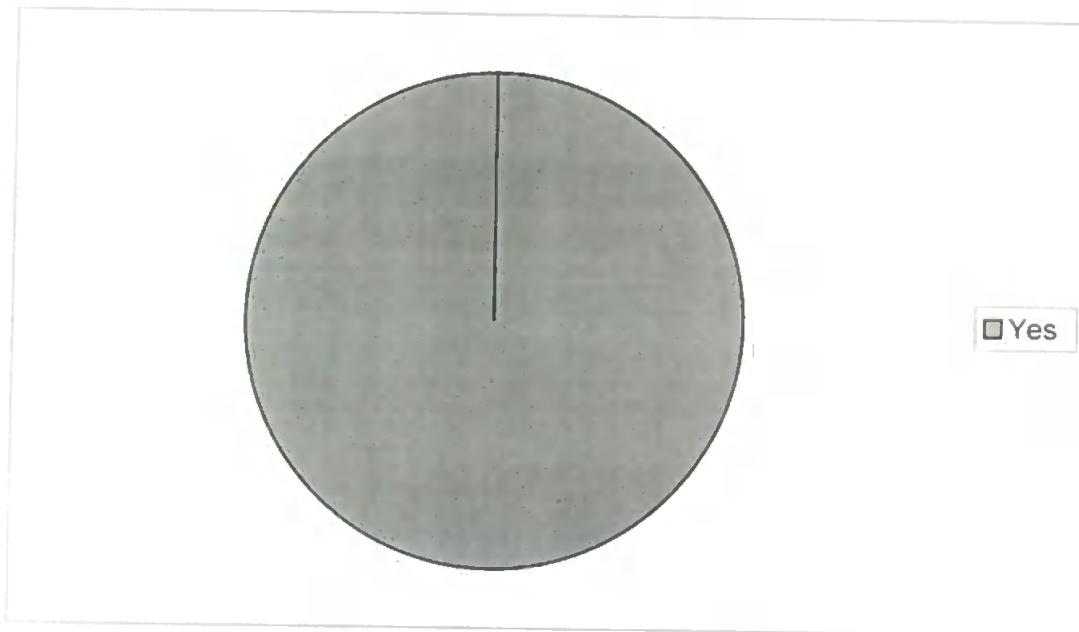


Figure 7-12 - Would it be useful to learn from the experience of others

Another significant result as it was a unanimous response – the experience of others is valuable in the learning process. However, this response does pose the question:

"How can experience be represented in order to communicate it to learners?"

This question is considered in greater detail in the following chapter.

13. How long did it take before you felt comfortable with component technologies?

Statistic	Value (months)
Min	1
Max	36
Mean	11.28571
Std. Dev.	9.278601

Table 7-8 - Time taken to be comfortable with component technologies

It is interesting to note that a lot of respondents who felt they were comfortable with component technologies in a short time were of "Intermediate" experience. This might suggest that they have not yet tried to exploit the most advanced features. On the other hand, a lot of experienced and very experienced developers had a period before they felt comfortable with the technologies that lasted beyond the mean response value. Of the four respondents who stated that they were still not fully comfortable with component techniques, one was intermediate, two were experienced and one was very experienced.

7.4.3 Regarding component technologies and the software development process

14. Was the integration of component-orientation into your development process straightforward?

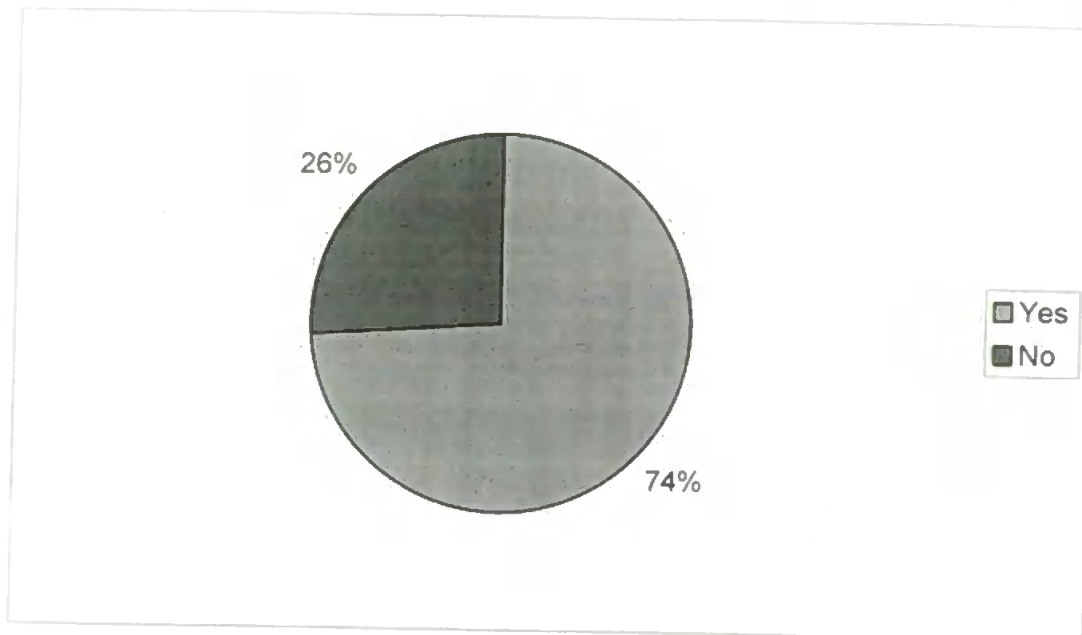


Figure 7-13 - Was integration straightforward?

While the majority of responses stated that integration was a straightforward process, the number of negative responses is significant. Certainly, it demonstrates that the experiences of the DOLMEN project are not entirely isolated. Further examination of response based upon both experience and types of component technology used highlight no additional patterns in integration (i.e. there was not a specific subset of respondents who experienced problems). However, an interesting comparison is to consider whether those who had problems with integration also experienced problems learning about component techniques.

Many respondents highlighted problems with the technologies themselves. Additionally, comments were made relating to organisational and personnel issues. Problems also listed included issues with interface definition version control and lack of consideration of the development process.

15. Do you believe that component-orientation makes software development: (harder, easier, neither easier or harder)

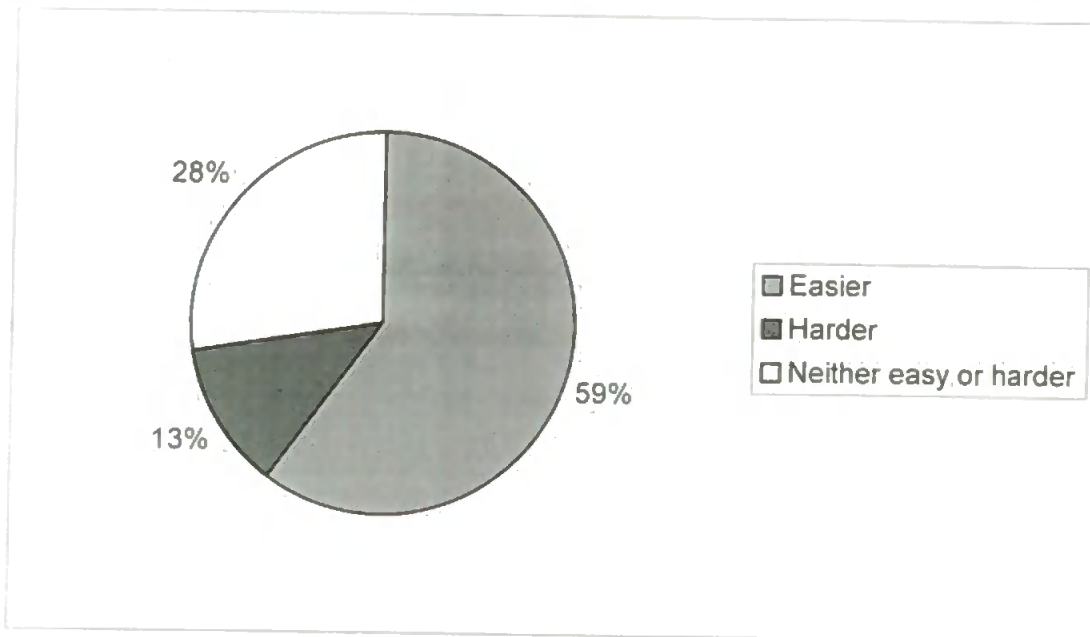


Figure 7-14 – Component-orientation makes software [easier, harder, neither easier or harder]?

The differences in response between the difficulties in learning but the lack of problems with integration and use developed from the responses in questions 14 & 15 seem to indicate that the main difficulties in the adoption and use of component technologies lie in the initial learning of concepts. However, it is still worth noting that while the majority of respondents felt that component-orientation made software development “easier”, the combined total of “harder” or “neither easier or harder” comes to 41%.

A comparison of responses against both technologies used and experience classification did not highlight any significant correlation.

Comments as to why component-orientation makes software development hard focused upon the complexity of implementation and the level of knowledge required to exploit a component standard and services – this relates directly to the issues in being a component *producer*, rather than *consumer*.

Another comment related to complexity in the organisation – all personnel involved in the software development effort have to be familiar with terminology and technologies in order that component-orientation is used effectively. We return to the issue of common language in the learning of a technique with our examination of learning approaches in chapter 8.

One particularly interesting opinion related to a perceived strength of component-orientation – black box reuse. The comment followed on from criticism of the lack of component documentation – in the case of source code reuse, this is not a major problem as the code itself can be used to determine dynamic behaviour. However, in the case of components, the interface and possibly some type information [64] are the only things that are available to the re-user in the absence of supporting documentation. The comment highlighted the value of source code in debugging, and the fact that third party reuse of component means that this is not available to the developer.

Respondents who stated that component-orientation made software development easier focussed primarily on the power of reuse that is afforded through binary objects. Many comments stated

that component-orientation meant that software development focused more on assembly and less on the coding of new functionality. Interfaces were also discussed as a means of making software development easier due to the contract between client and server – a client developer can work to the specified interface without having the implementation with them. A final positive aspect is seen in the network transparency that distributed standards provide, as this saves a good deal of low level programming in the development of bespoke network interfaces.

However, a few comments were more equivocal, returning to the importance of the learning process in gaining the benefits of component-orientation.

Comments from respondents who stated “neither easier or harder” seemed to provide a balance between the two views, with comments such as “easier to define, harder to implement”, and “easier for reuse but harder to learn”.

16. Given the choice, would you use component-oriented techniques when developing software (always, sometimes, occasionally, never).

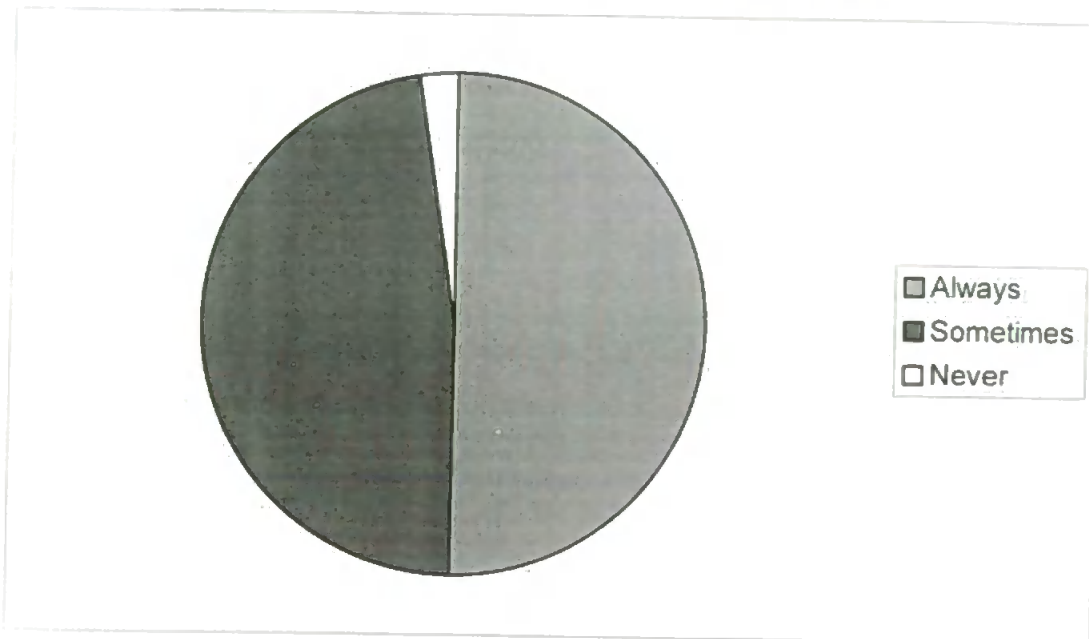


Figure 7-15 – Willingness to use component technologies

No trends emerge from comparison of responses against use of technology or level of experience. However, when comparing component use against opinion whether the technologies make software development harder or easier, there is an interesting result (see Figure 7-16).

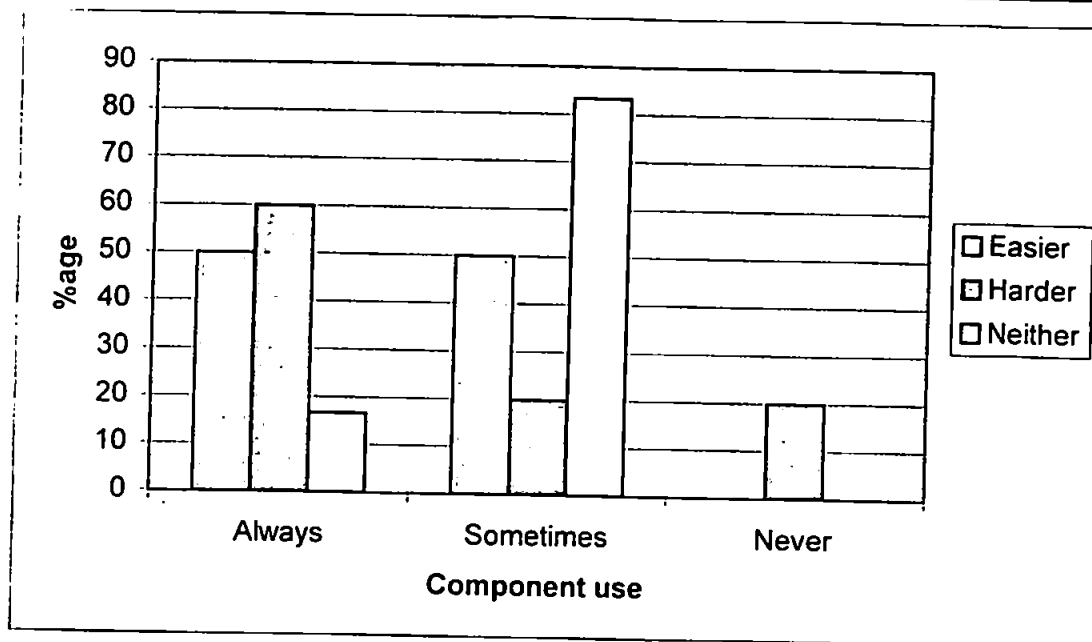


Figure 7-16 - Comparison of component use with opinion regarding component difficulty

As the figure shows, among those respondents who considered that component-orientation makes software development harder, 60% would still use component techniques all of the time for future development projects. It also shows that those respondents who thought that component-orientation made software development neither easier or harder were most cautious with its use, the vast majority saying that would use component-orientation only sometimes.

This response can be compared with the Netscient case study, where components were used to the strength in the distribution and sharing of functionality, but for other aspects of in house systems, they were not considered appropriate and other techniques were used.

17. Component-orientation is easily adopted into the development process

This question was posed because the DOLMEN case study seemed to demonstrate that adopting component-orientation into a development process was problematic. While the Netscient case

suffered far fewer problems, we could not test whether this was due to “lessons learned” from the DOLMEN case or a more normal experience in the adoption of component technologies. The results from the survey would suggest that the DOLMEN case experience was not the norm and that component technologies can be adopted in a straightforward manner. This leads to asking the question:

Why did adoption go so badly wrong in DOLMEN?

Firstly, the DOLMEN development process itself comes into question. It has been criticised in the case study analysis as being too rigid and linear to be able to both review progress and also adapt to unexpected occurrences that could arise from the use of new technologies. Additionally, little provision was made for the familiarisation or learning of the new technologies before commencing development. The case studies have demonstrated the complexity of component technologies and the importance of being aware of both their strengths and weaknesses before using them in development projects. Survey responses that have highlighted the problems with the learning of component technologies also contribute to the argument for having good knowledge of them before commencing development.

It should also be noted that while the majority response for this question has been positive, there is still a fair proportion of respondents who do not believe that component technologies are easily adopted into the development process. Therefore, while the DOLMEN experiences are certainly in the minority, they are by no means unique.

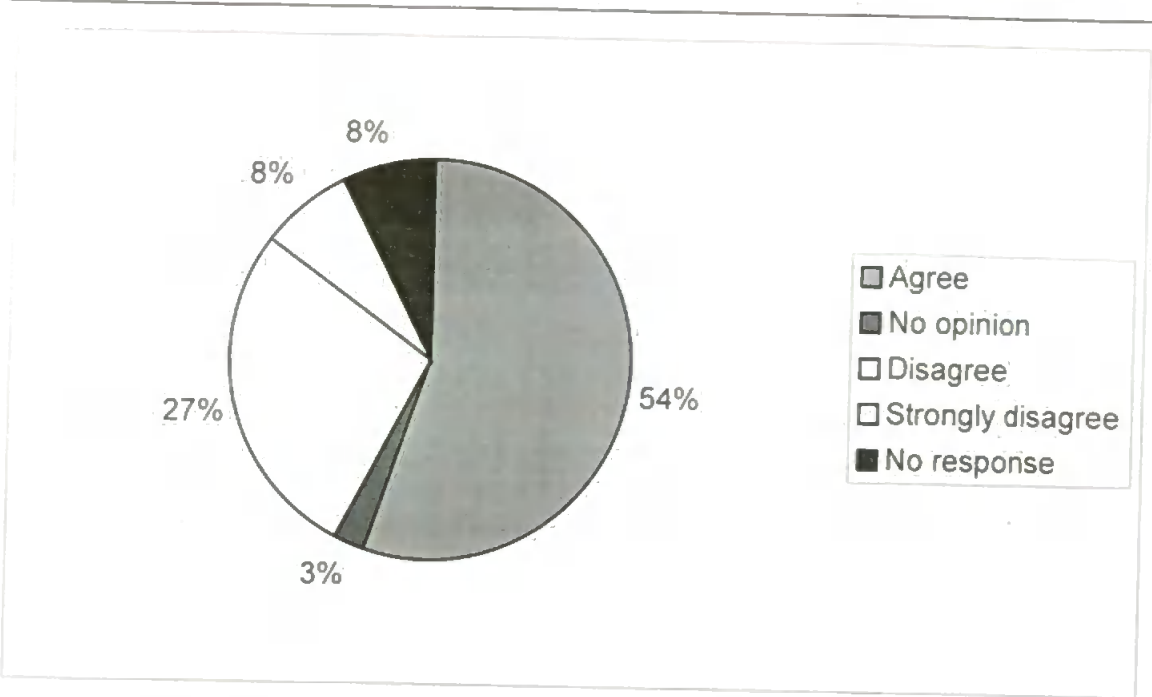


Figure 7-17 - Component technologies can be easily adopted

While the survey provided little evidence of any difference due to the type of component technology used, an interesting outcome can be seen from comparing responses in this question to those people who had problems integrating the technologies themselves. Figure 7-18 illustrates the difference in response between the respondents who did experience problems, and those who did not. The most surprising thing about this comparison is that almost exactly the same proportions in each group provided similar opinions. One would expect those who experienced problems integrating component technologies into their own development processes to feel that they are not easy to adopt. However, the results presented here do not confirm this expectation.

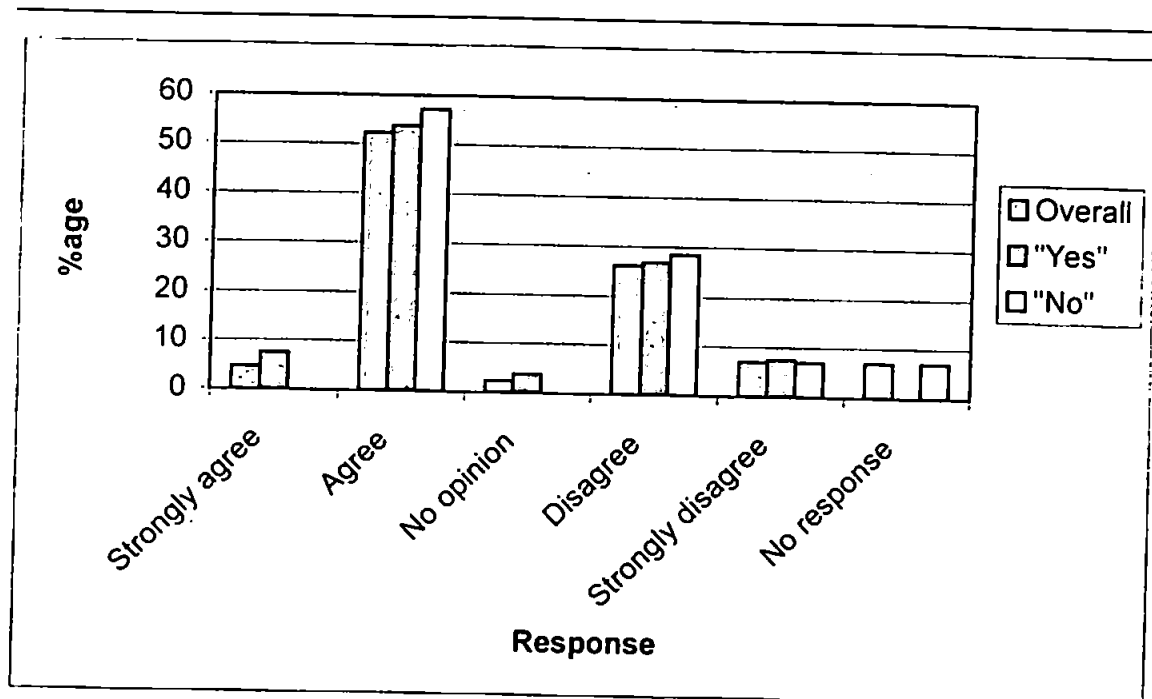


Figure 7-18 - Ease of adoption vs. integration problems

18. Component technologies can be adopted independently of wider organisational consideration

There is a more or less equal split in the responses here between those who agreed or strongly agreed with the statement, and those who disagreed or strongly disagreed. If the survey respondents reflected case study findings, we would expect those who agreed with the question to have experienced problems with adoption and use (as occurred in the DOLMEN project). Conversely, while those who disagreed had a far more straightforward adoption (as occurred in the Netscient project). The survey responses showed no such patterns.

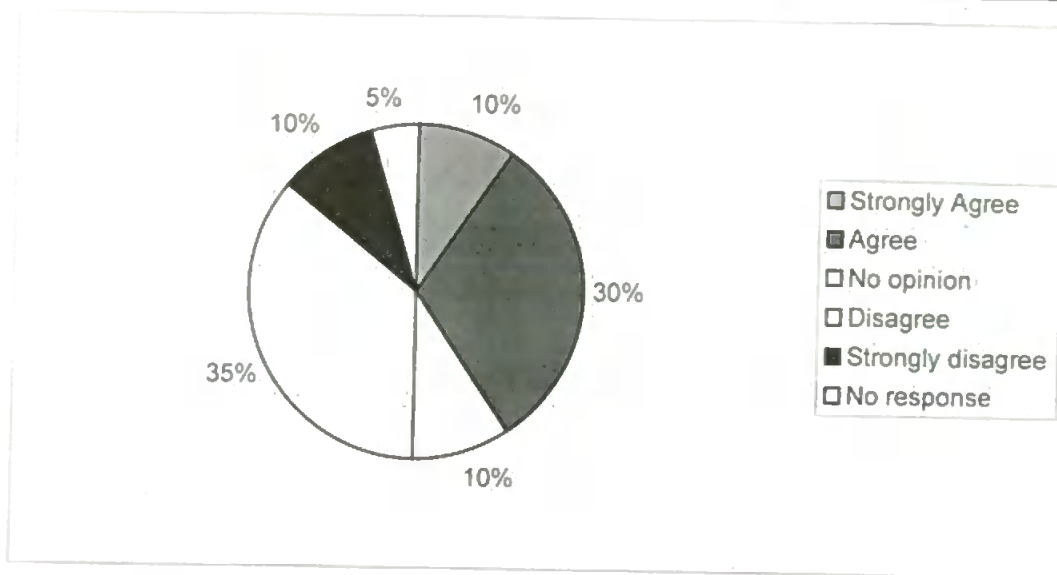


Figure 7-19 - Component technologies can be adopted independent of organisation issues

However, a comparison of opinion against experience of component technologies (see Figure 7-20) does indicate a difference in opinion based upon technology type. A large number of COM-only respondents thought that component techniques could be used independent of wider issues, while respondents who had used both technologies or just CORBA felt, in the majority, that they could not. This may be because CORBA still tends to be seen as an extension of the base platform on which to develop software. While some platforms are starting to incorporate CORBA (such as GNU's GNOME project – <http://www.gnome.org>), in the vast majority of cases, a CORBA implementation and third party CORBA objects have to be bought in.

The Windows platform provides COM as a standard subset and any Windows system will contain countless COM components – it is implicit in Windows development. Additionally, the ease in which a developer can become a component consumer on the Windows platform – through the use of tools such as Visual Basic, means that it does not seem like such an undertaking to start using COM technologies. The perception is that CORBA requires real understanding and

commitment whereas a project can exploit a few existing COM components can be done with little extra effort.

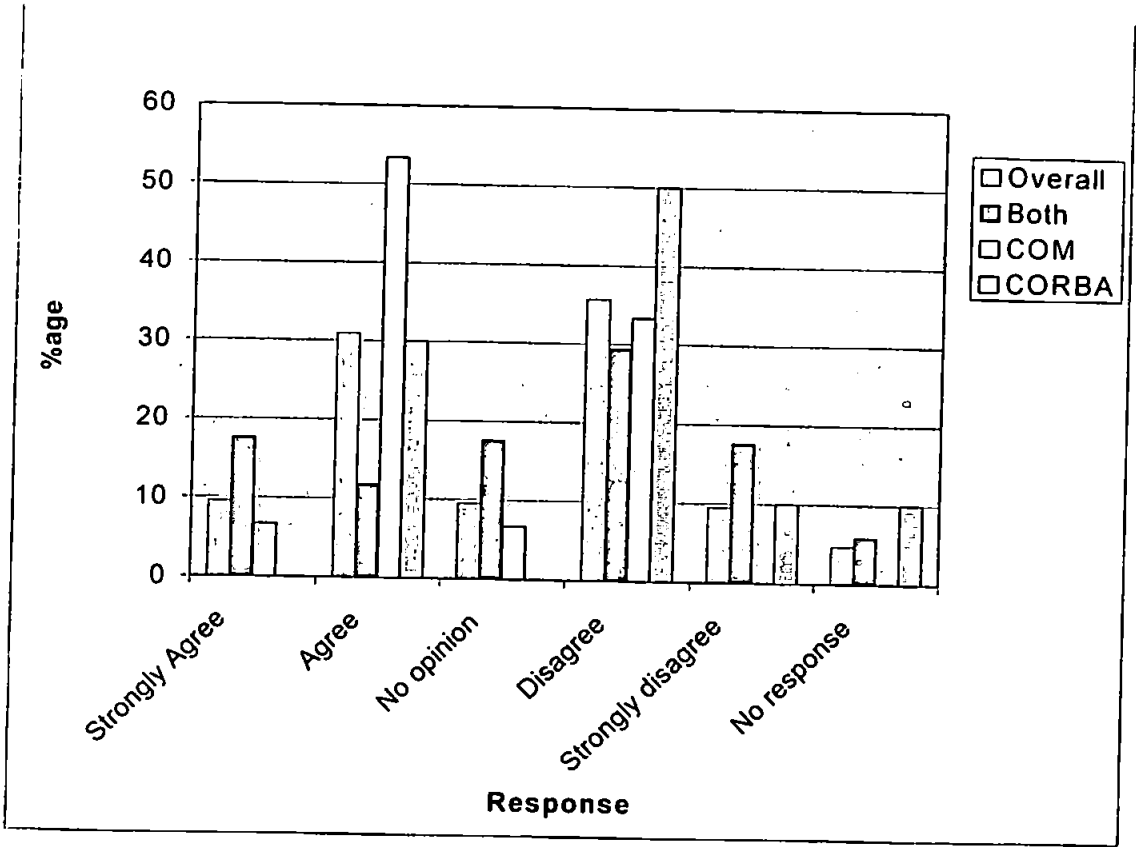


Figure 7-20 - Comparing agreement with question 18 against use of technology

19. Project management is unaffected by component technologies

Figure 7-21 demonstrates a very strong response disagreeing with the statement presented in the questionnaire. It confirms one of the issues arising from the DOLMEN case, where component orientation was considered to be an implementation technology that was not of concern for the project management. This response greatly strengthens the opinion that this approach to the use of component technologies was wrong, and that project managers need to be aware of the issues in their use as much as developers.

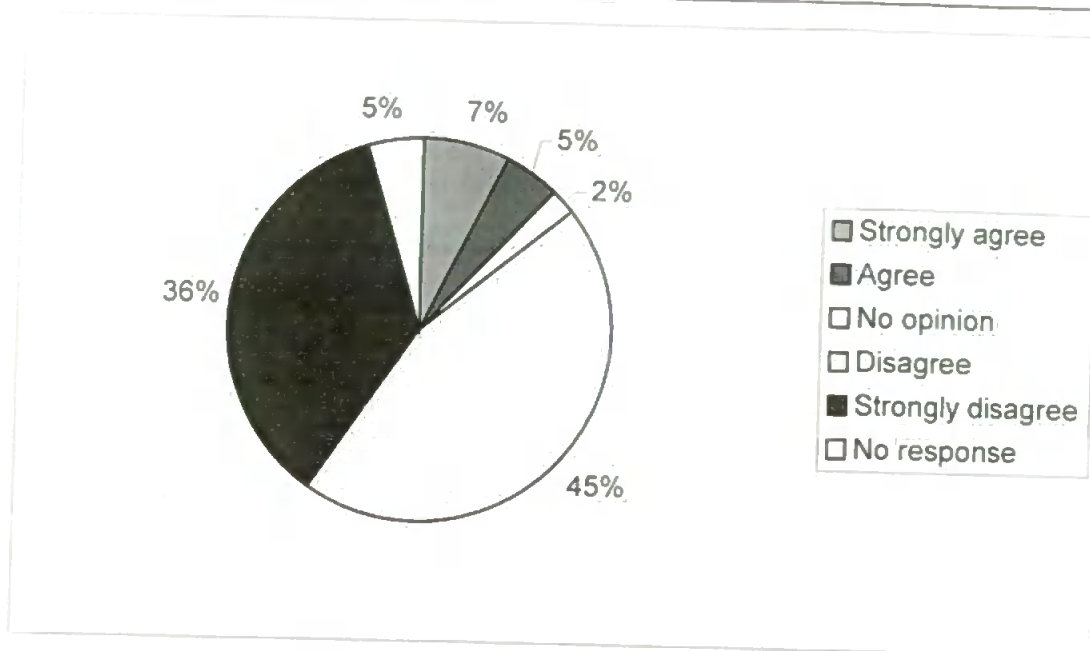


Figure 7-21 - Project management is unaffected by component technologies

20. Component-orientation makes software reuse easy

One of the underlying philosophies of component orientation, ever since its theoretic introduction by McIllroy [100] is that it makes software reuse possible on an industrial scale. Industry literature (for example [37], [33]) is especially keen on the reuse aspect of component orientation. The case studies had experienced mixed results in generating large-scale reuse: DOLMEN had not been at all successful in developing reusable components, whereas Netscient was. The analysis of the case studies suggested that it was perhaps not the technologies themselves, but their use in DOLMEN that hampered reusability. The response from respondents in the survey (see Figure 7-22) would also indicate that the DOLMEN experience was not typical - the majority of respondents either agreed or agreed strongly with the statement.

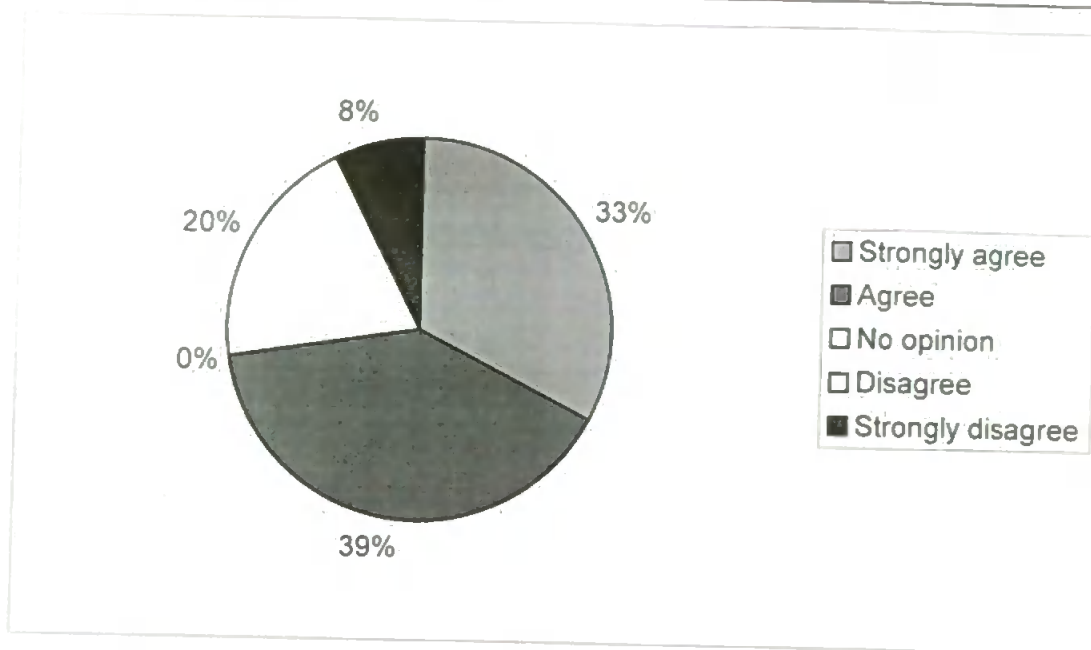


Figure 7-22 - Component orientation makes software reuse easy

However, a significant proportion (28% in total) that either disagreed or strongly disagreed. This promoted an examination of responses against the type of technologies used (as DOLMEN had been a CORBA oriented project, compared to Netscient's COM approach). As can be seen from Figure 7-23 – CORBA only respondents accounted for most of the negative responses, while COM only respondents were largely positive.

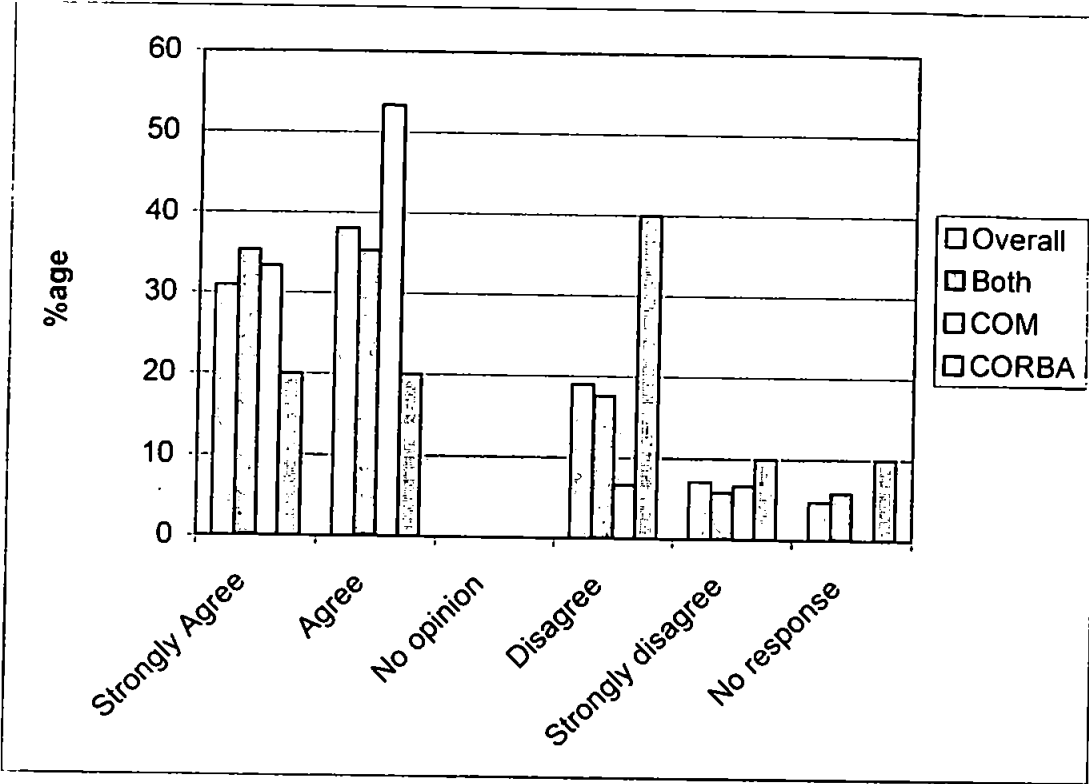


Figure 7-23 - Ease of reuse compared to technology experience

21. Component-orientation should focus upon software reuse

This question was intended to complement the previous one, asking whether component orientation *should* be reuse focused. Figure 7-24 shows another positive response, with the majority of respondents of the opinion that component orientation should be reuse focused.

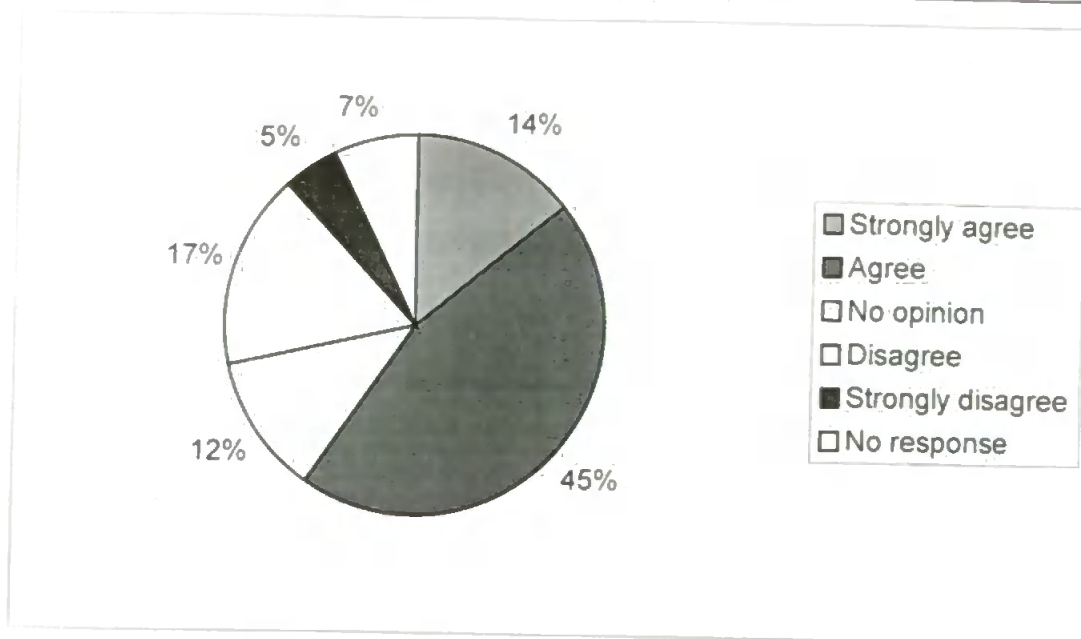


Figure 7-24 - Component orientation should focus on software reuse

In this case, there was no great difference between respondents based upon technology experience. Perhaps this suggests that while CORBA developers find that software reuse is difficult to achieve, it should still be one of the drivers in using component technologies.

22. Using component technologies is straightforward

This question relates to the complexity of component technologies, in the view of practitioners who have used them. This, in turn, impacts upon their adoption into the mainstream (as discussed elsewhere in this thesis – see section 3.2). The interest arises in comparison with some of the more positive responses. For example, the answers to questions 17 (Adoption is straightforward) and 20 (reuse is easy). One might assume that those positive outcomes signal the ease of use of component technologies. However, the fact that the majority response was to the contrary suggests once again that it is only when developers are fully aware of issues in the use of technologies that they become truly easy to use.

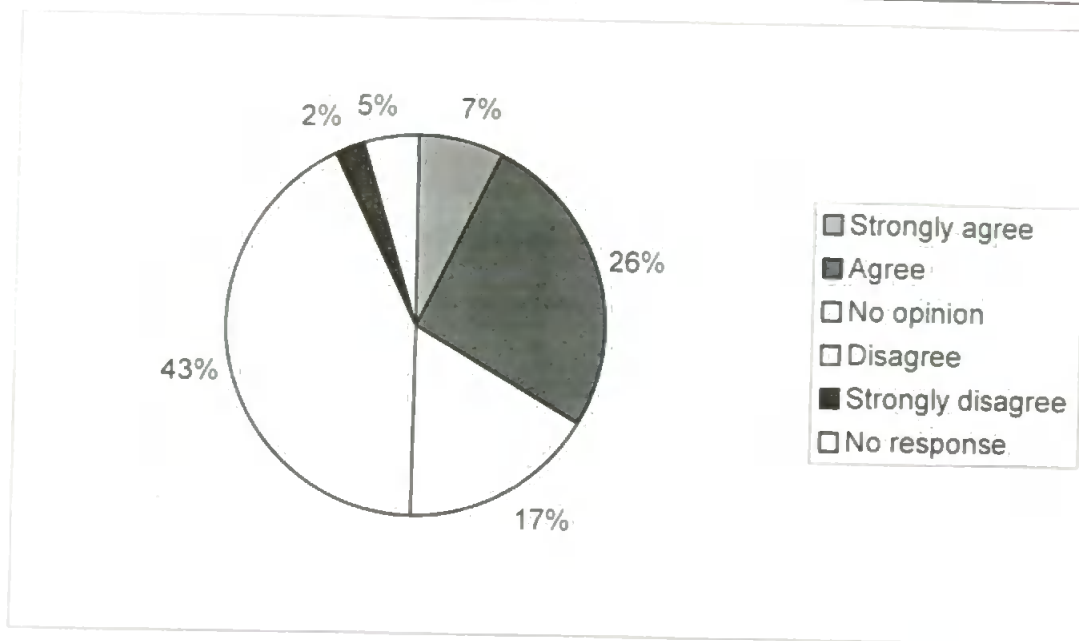


Figure 7-25 – Using Component Technologies is Straightforward

23. A component-oriented approach encourages design

The positive response illustrated in Figure 7-26 confirms an unexpected outcome from the Netscient case. While it was not an explicit part of the investigation, it became apparent, through both direct and participant observation, that in order to keep track of the mix of component clients, in house components, third party components and external systems, design documentation was being used to far greater effect than in previous non-component-oriented projects within the organisation. As this was an unplanned outcome that, in essence, produced the hypothesis "Component-orientation encourages design activities", it was important to test this. It emerged that 83% either agreed or strongly agreed with such a statement.

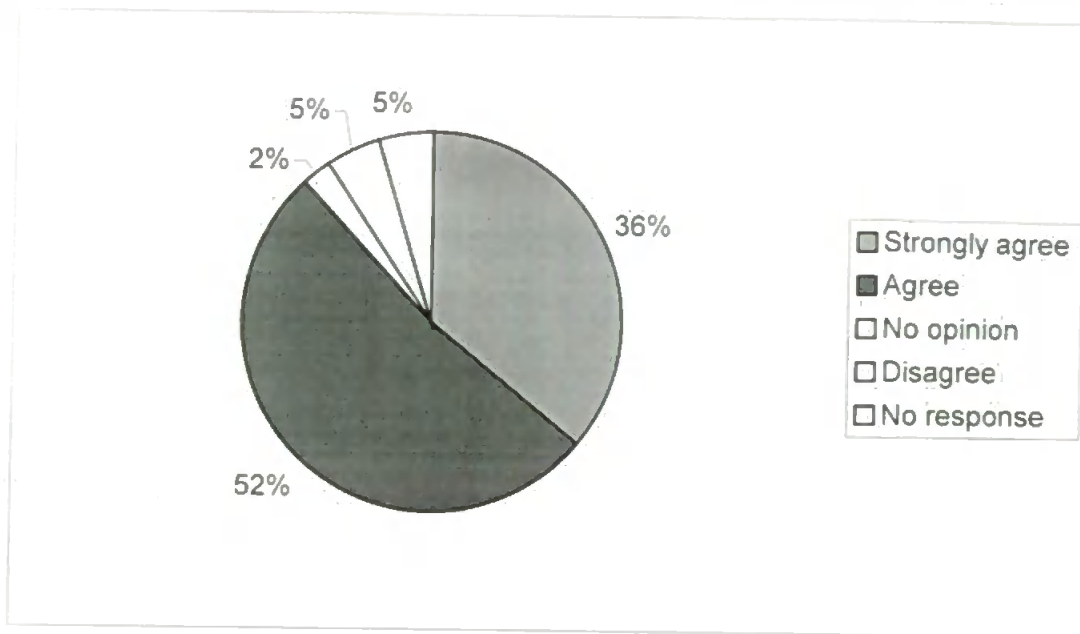


Figure 7-26 - Component orientation encourages design

24. Component based development makes system deployment easier

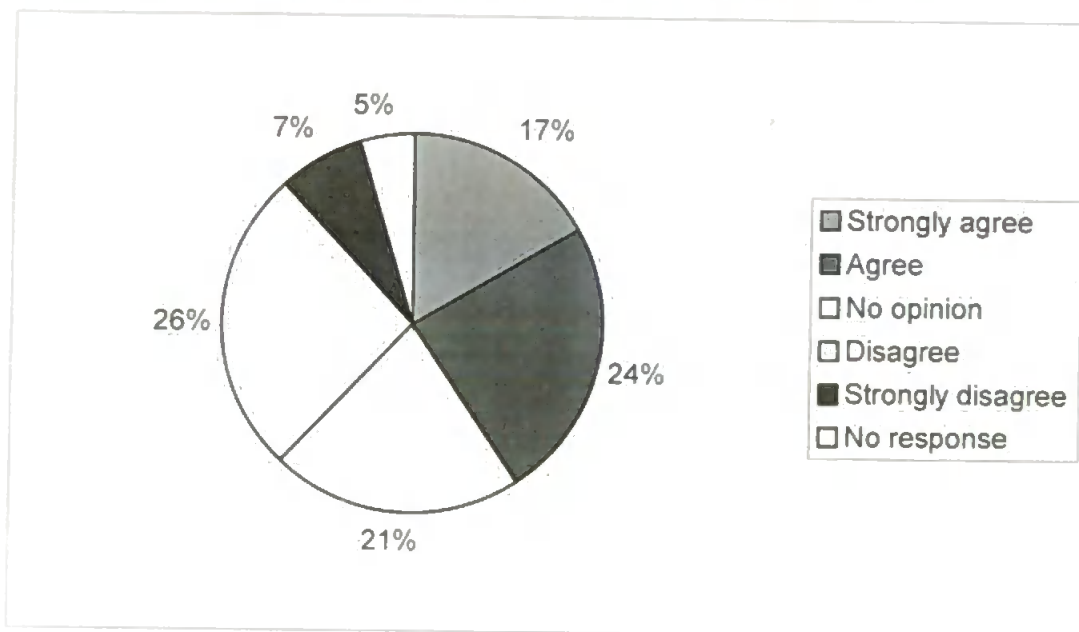


Figure 7-27 - Component based development makes system deployment easier

A comparison of opinion against technology experience (see Figure 7-28) again highlights significant correlation. It seems that CORBA-only respondents have a majority response agreeing with the question whereas COM-only developers find deployment a more complex task. However, CORBA only respondents also have a response that is above the overall response in disagreeing with the statement. Only respondents experienced in both CORBA and COM development are below the overall response in disagreeing with the statement. It is also interesting to note that respondents who have used both types of technology have the greatest response of "no opinion". This suggests that component orientation either has no effect upon deployment or its complexities and benefits balance each other out. An opinion that might be thought to follow from this response – that the more experienced developers are the ones who have seen both good and bad points in deploying component system – is not supported by a comparison of response against experience.

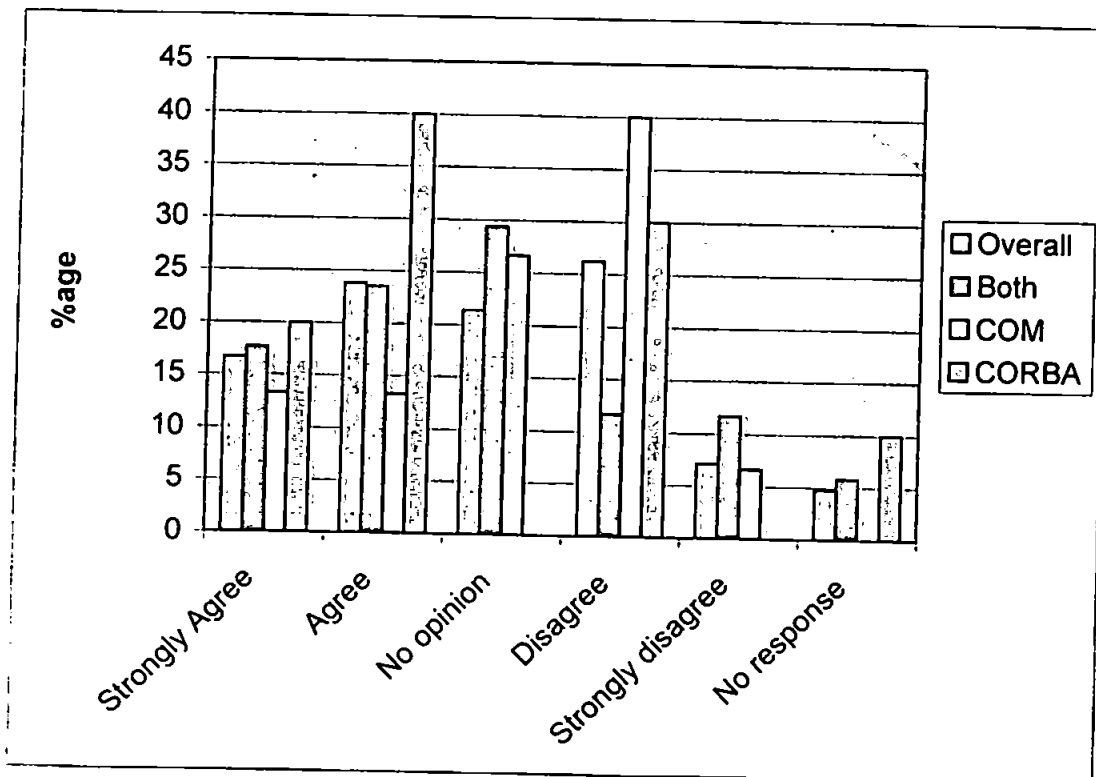


Figure 7-28 - Ease of deployment against technologies used

25. Component based development makes system maintenance easier

This final question again tests experience of technology against underlying philosophy (see section 2.6). Another purported strength of component orientation is that it eases system maintenance. Theoretically, the use of interfaces, black box and binary reuse means that a component can be bug-fixed and plugged into a live system without any component clients needing to be brought down in the maintenance (for example, see [33]). As this issue could not be tested in either of the case studies (as, in each case, they were only studied until the first version release of the software), this final question was used simply as a test of practitioner experience. It would seem, given the positive responses to the question that this aspect of component orientation is borne out by practitioner experience.

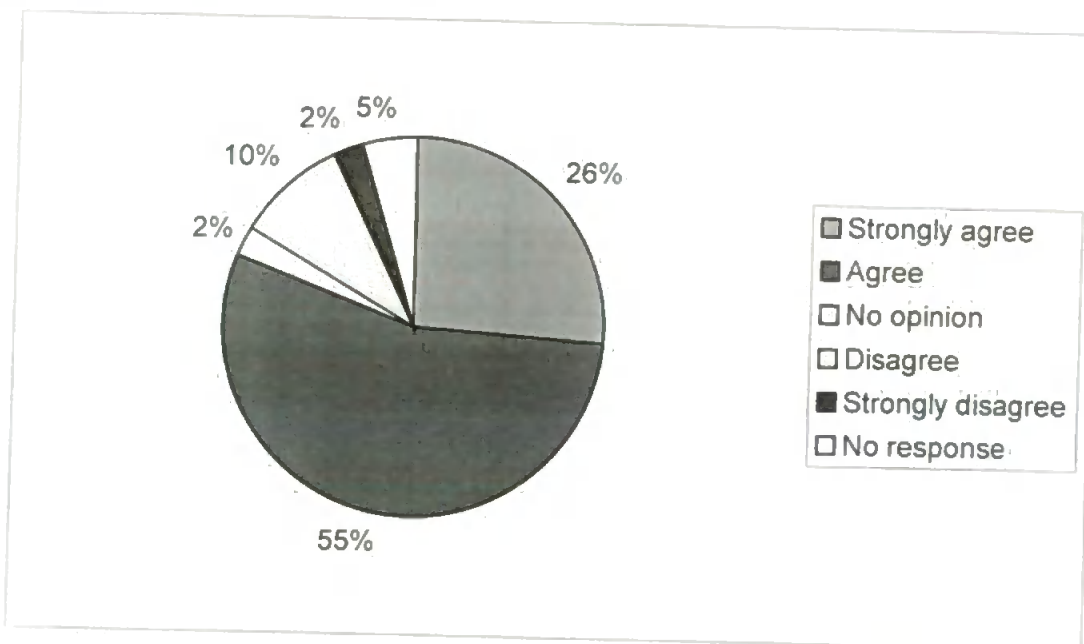


Figure 7-29 - Component orientation makes system maintenance easier

7.5 Implications of Survey Results on Case Study Findings

7.5.1 Case personnel responses

The following compares personnel responses from both cases against model/majority responses from the total survey respondents in relation to the final section of the questionnaire related to experience of component technologies' effect on the software development process. In each case study two respondents were asked to carry out the survey, so their experiences could be measured directly against the experience of others.

The most surprising aspect of this comparison is the amount of agreement between, in particular, the DOLMEN respondents but also the Netscient respondents, and the majority response. Conflicts with the majority responses are highlighted in the table, illustrating the number of responses in agreement. This comparison goes some way to confirm that the experiences of personnel in both case studies are similar to the experiences of others within the field. This is encouraging in determining the external validity of the case studies, as a common criticism of case study research is the problem of generalisation of results [167].

	<i>Model respondent</i>	DOLMEN 1	DOLMEN 2	Netscient 1	Netscient 2
14. Straightforward integration	Yes	No	No	No	No
15. Development easier or harder	Easier	Harder	Harder	Easier	Easier
16. Use component technologies	Always	Sometimes	Never	Sometimes	Always
17. Easily adopted	Agree	Disagree	Strongly disagree	Disagree	Disagree
18. Independent adoption	Disagree	Disagree	Disagree	Disagree	Disagree
19. Project management is unaffected	Disagree	Strongly disagree	Disagree	Strongly disagree	Disagree
20. Reuse is easy	Agree	Agree	Agree	Agree	Agree
21. Focus upon reuse	Agree	Agree	Agree	Agree	Disagree
22. Easy to use	Disagree	Disagree	Disagree	Disagree	Disagree
23. Encourage design	Agree	Agree	Agree	Agree	Strongly agree
24. Make deployment easier	Disagree	Disagree	Disagree	Disagree	Strongly agree
25. Make maintenance easier	Agree	Disagree	Disagree	Agree	Strongly agree

Table 7-9 - Comparison of model respondent against DOLMEN and Netscient

respondents

7.5.2 Comparison of Responses Against Case Study Propositions

7.5.2.1.1 General Case Propositions

- 1. Adopting and using component technologies in software development processes will affect process activities**

There are some very positive responses in the survey that strengthen this proposition. In particular questions related to project management, system design, deployment and maintenance (see questions 20, 24, 25 and 26 respectively) all resulted in responses that would confirm the effect the component-orientation has on development activities.

- 2. An awareness of the issues involved in the adoption and use of component technologies can ease their integration**

The major theme that runs through responses in this survey reflects the fact that learning and understanding of component technologies is *the* issue in using them successfully (in particular, see discussion regarding questions 16, 17, 18 and 23). Therefore, this proposition has been greatly strengthened by survey results.

7.5.2.1.2 DOLMEN Case Propositions

- 3. Component technologies ease the development, integration and deployment of distributed systems**

The distributed aspect of component-based development was not explicitly addressed in the survey, but positive responses to questions such as 25 and in particular the discussion regarding questions 16 and 17 highlight the fact that component technologies can be used to address the low level elements of distributed development. Indeed, one respondent to question 16 explicitly stated the benefits of using component software rather than having to hand craft network interfaces.

4. Uncontrolled adoption and use of component technologies can have a negative affect upon a development project

Drawing from the central outcome of the survey relating to the need for understanding, the proposition is demonstrated to have some validity. Undoubtedly, the experiences of the DOLMEN project are very much in the minority among component practitioners. They are not, however, unique. This in itself strengthens the issues identified in the DOLMEN case study as possible outcomes when using component technologies, if such use is not carefully considered.

7.5.2.1.3 Netscient Case Propositions

5. A domain-oriented approach to component development provides a greater degree of reuse than a product-oriented view.

This proposition was not explicitly addressed in the survey. However, responses to questions related to software reuse issues through the use of component technologies would suggest that the level of reuse achieved in the Netscient case is not uncommon. Therefore, it would seem that the lack of reuse potential shown in the DOLMEN case would once more relate to a lack of understanding of the component technologies, rather than an inherent problem with them.

6. Similar issues with component-orientation occur when using different technologies from the same field (i.e. Microsoft based, rather than OMG based technologies)

Several questions have highlighted differences in experience relating to the types of technologies used by respondents, in particular, questions 19 ("Component technologies can be adopted independently of wider organisational considerations"), 21 ("Component-orientation makes software reuse easy") and 25 ("Component based development makes systems deployment easier"). However, we cannot illustrate any explicit trends throughout the survey (i.e. there is

nothing to suggest that CORBA will always result in poor development, whereas COM will always results in effective development). Therefore, once again, were are drawn back to the issue of front-loading knowledge when using component-oriented techniques – with an awareness of the issues and an understanding of the technologies, effective development can be achieved, regardless of their type.

7. Issues in the DOLMEN case study can be avoided through greater knowledge of the technologies involved

Relating back to the issue discussed in proposition 2, it has certainly been illustrated in the case study that awareness and understanding are the important issues in using component technologies.

7.6 Chapter summary

In order to address issues regarding the external validity of research findings from the two case studies in relation to the effect of component-orientation upon software development, a practitioner survey was carried out. This chapter has reviewed the method used in carrying out the survey and considered the responses obtained against case study findings and their development. The survey has highlighted issues in the adoption and use of component technologies that are particularly problematic and also enabled further examination of case study propositions based upon the experience of others. This, in turn, allows the determination of those issues that should drive the development of research results.

The survey highlighted the problematic areas as learning, and understanding of the issues involved in component-based development. The following chapter examines how these issues can

be addressed and discusses a strategy for development based upon both research findings and also and examination of relevant literature.

This chapter proceeds from the assessment of component technologies to consider the most effective approach in developing results to a form suitable for adopters. An effective approach should consider adoption theories, the way that an organisation learns and the nature of the results from the research programme. In addressing these issues, an effective strategy for the transfer of research experience to organisational knowledge about component technologies is developed.

8. Adopting and Using Component Technologies

This chapter examines the development of results from the research analysis into a form that can be used by practitioners wishing to learn from our research findings. It begins by considering the nature of results from the case studies and survey, and discusses problems in their presentation as a learning aid. In examining the development of the results, two new research aims are presented. The remainder of the chapter is divided into two distinct parts. Firstly, the suitability of a reference model for component platforms is discussed, drawing from literature related to the use of such models in software engineering. It is argued that the reference model not only enables a comparison of similar technologies but can also be used to aid in the construction of knowledge regarding component technologies, exploiting the implementation independent nature of such models. The reference model is defined, and its use is demonstrated against existing platforms.

The second part of the chapter concerns the further development of results in order to encapsulate experience into a learning strategy. Drawing from previous discussion related to organisational learning, this section examines existing approaches for technology transition and discusses the suitability of such approaches for our own needs. Pattern approaches are examined in further detail and reasons for their suitability to our own requirements is discussed.

The chapter ends by defining a strategy for the sharing of experience in the adoption and use of component technologies, drawing together both the reference model and patterns approach in specifying a contextualised pattern language that serves as a learning tool for practitioners.

8.1 *Developing Case Study and Survey Results*

The results from the case studies and survey provide an assessment of the impact of component technologies upon software development. However, the aims of the research include a requirement to provide a tool for practitioners to learn from the findings of this research programme. Additionally, the survey highlighted the desire of practitioners to learn from the experience of others within the field. The results from the analysis methods provide a number of theories related to the adoption and use of component technologies, and a validation of those theories. However, the results are not suitable, in their current form, as a learning tool.

In developing them it is important to keep their essence while restructuring them to provide an effective representation of experience usable by learners - any outcomes should be of use to an organisation wishing to bring component-orientation into their own development approach.

This raises the issues:

1. how does one communicate experience to a learning organisation?
2. how can the results be best developed to fit into the learning process?

We first examine the role of reference models in communicating concepts. In developing a model for component platforms, we aim to address a learning need in reconciling concepts to implementations in component technologies and provide a tool to aid further development of the above issues.

8.2 A Reference Model for Component Platforms

8.2.1 Component Platforms

An element of the case studies that enabled the placing of each into context was the definition of the platform used to develop software. It defined component infrastructure, services used and the nature of components developed on top of the infrastructure. In the case of Netscient, it also related other software technologies to provide a definition of the related technologies used in the development of software. Additionally, such a platform definition enabled a direct comparison of the development technologies used in each case, proving an aid to reliability for the cases [166].

The following develops the concept of component platforms in a more generic sense.

8.2.2 A Reference Model for Component Platforms

The value of reference models in software engineering is to provide the means to compare different systems within a domain. Sommerville [145] identifies this value and discusses the use of models in both the networking domains and the software development environment domain. Rine & Nada [131] also discuss the value of reference models in software engineering, defining a model for software reuse that is used to compare the reuse strategies of a number of organisations. Well known reference models, such as the OSI seven-layer model for networking [76] define aspects of the domain. Another reference model that is related more closely to this research programme is the Reference Model for Open Distributed Processing (RM-ODP [78]). This model identified a need for co-ordination in the definition of distributed processing, responding to a rapid growth in the domain and confusion related concepts and implementations. The model defines common aspects of distributed processing, such as distribution, interoperability and portability, without relating any aspect to a specific implementation. The model has been influential in the field, and has been used by, among others, the OMG for the

CORBA standard. It was also used for initial architectural consideration in the DOLMEN project (see section 5.1).

Another important feature of reference models is that they provide an implementation independent view of a specific domain. Our work in the domain of component-orientation has highlighted problems in the reconciliation of concepts with implementation technologies. This issue was initially identified in the DOLMEN case study, and amplified through the practitioner survey. Many responses commented upon problems with definition and vocabulary, and also relating theory to implementation. In developing the results from the research in this programme, we aim to provide assistance in the learning and adoption of component technologies. A model that enables the definition and comparison of component platforms is a valuable tool for the learner. This opinion is in line with other approaches to technology adoption – the guidelines for the development of transition packages within the SEI's Transition Enabling Program [57] recommend the use of reference models within a transition package.

The following expands on a simple model presented in [68] in defining a reference model for component architectures. Hoffmann's model drew on his own work in the development of concepts in component-orientation, and aimed to distinguish between essential elements within component-based development. The model presented a simple distinction between components, component infrastructure and distribution infrastructure and is illustrated in Figure 8-1. The term "component platform" is used in preference to "component architecture" partly to distance the model from the wider research area of software architecture [142], which relates to the structure and design software system. Also, as the model defines the infrastructure on which systems are built, the term platform is more suitable. Figure 8-2 illustrates the reference model, which

incorporates greater details into the aspects of a platform, as well as differentiating between essential and non-essential features.

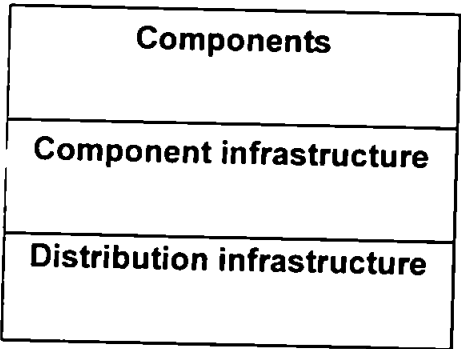


Figure 8-1 – Original Component Architecture Reference Model taken from [68].

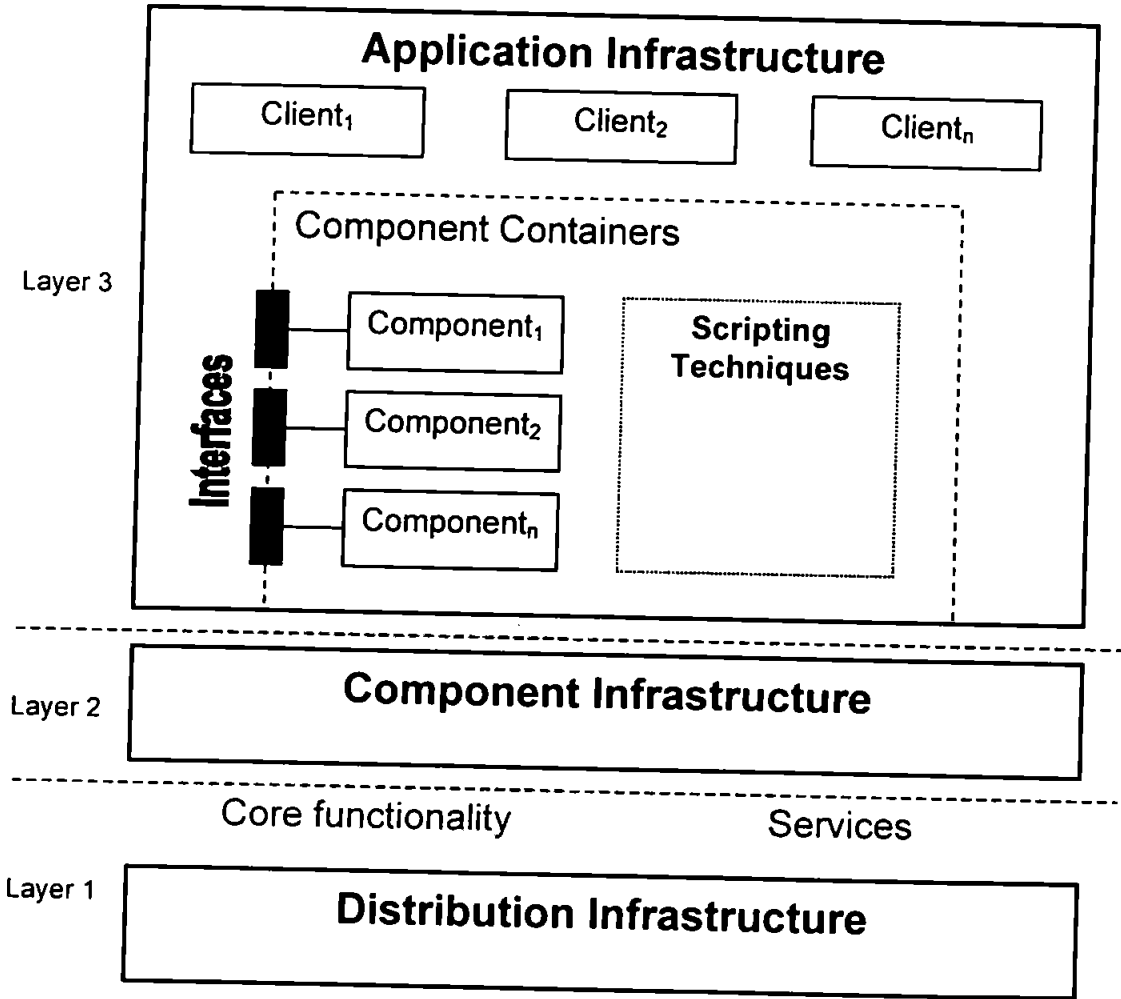


Figure 8-2- Reference Model of a Component Platform

8.2.2.1 Overview

Layers 1 & 2 are broken into two parts, namely core functionality and services. The core functionality of an infrastructure provides what is required to enable any component construction and interaction. Component services add value to the functionality, perhaps implementing a function that would otherwise need to be created by the developer (for example, security, transaction control, licensing, etc.). It is possible for a platform to be free of these services, but it would be far harder to develop componentware without them. On the other hand, it may not be necessary to provide a huge suite of component services to complement the standard.

8.2.2.2 Layer 1 – Distribution Infrastructure

The distribution infrastructure defines the protocols and services required enabling the platform to provide distribution and network transparency for any system developed on the platform. Core functionality should define:

- **Network protocols:** The protocols by which inter-component and client/component interaction can take place.
- **Marshalling/unmarshalling:** The way in which a component call is packed, transmitted, received and unpacked across the network.

Additional services enable a more efficient distributed implementation by providing functionality for core services within distributed applications, such as:

- **Security:** Providing functionality to ensure secure communication between clients and components and also between components. Such functionality could include authentication of calls, non-repudiation, etc., for example, the CORBASecurity service [110].
- **Distributed naming and location:** A central or distributed repository of component names that enables client and other components to call component functionality without knowing the component's location on the network.

8.2.2.3 Layer 2 – Component Infrastructure

The component infrastructure offers protocols and mechanisms for inter-component interaction, component structuring, etc. as well as component services (e.g. versioning, licensing, monitoring, etc.). Core functionality should include:

- **Component structuring:** Defining the structure of the component (properties, methods, events, etc.) and the way the component exposes its structure to clients.
- **Inter-component interaction:** The protocols used in calls from clients to components and between components.

Component services can be extremely varied, but all aim at making the development of component based systems more straightforward by implementing common functionality within such systems. They may include:

- **Licensing:** Providing some level of control over who can use the component.
- **Monitoring:** Enabling an external client to monitor a component and respond to component behaviour (for example in an event based system)
- **Scaling:** Providing the means to transparently scale the component's use from single to multiple clients, without the component developer needing to be concerned with such issues as threading, resource control, etc.
- **Persistence:** Keep a component instance "alive" when not resident in memory so that properties of the instance can be recalled at a later time.
- **Transaction control:** Ensuring atomic transactions to counter the possibility of system crashes when writing to a database or similar.

The split between distribution infrastructure and component infrastructure is important: a component system could exist on a standalone machine with no network distribution. JavaBeans

and COM are examples of component standards providing functionality solely for stand-alone systems.

8.2.2.4 Layer 3- Application Infrastructure

The application infrastructure provides the immediate aspects of the component system that are available to the application developer in order to build software systems. It uses the distribution and component infrastructure as a foundation and develops components and component clients based upon this infrastructure in order to develop systems. The application infrastructure comprises of a number of elements:

- **Components:** Components themselves are the packaged, reusable binary units that implement a given aspect of functionality. They will be structured in line with the component infrastructure, exposing functionality via some form of interface, and are held in memory in some form of component container.
- **Interfaces:** Interfaces are the means by which a component exposes its functionality to the outside world. This is defined generically within the reference model, no specific technique is assumed for achieving this. However, CORBA and DCOM (see below) define component interfaces using an interface definition language.
- **Component containers:** Enable the execution of instances of a component within a component system by providing an environment in which they can be loading into memory. Containers can be independent units (for example, a DLL or an ActiveX control), or be part of an executable process (such as a CORBA server process).
- **Component clients:** Elements that will, in general satisfy user requirements through the assembly and scripting of components within the infrastructure. Such clients can be part of a component container itself (for example COM automation servers such as Excel), or be

external to the component infrastructure, accessing component functionality contained, for example, within DLLs.¹⁰

- **Scripting techniques:** These techniques enable easy assembly and calling of component, and also provide basic programming constructs such as loops and conditional statements in order to add simple custom functionality within a client. A familiar example of a scripting technique within a component infrastructure is Microsoft's Visual Basic for Applications.

8.2.3 Current Standard Component Platforms

The reference model can be used to compare platforms from the two case studies (see sections 5.4 and 6.5). To further demonstrate its use, the following applies it to two major component platforms: the OMG Object Management Architecture (OMA) [109] and the Microsoft Distributed interNet Application Architecture (DNA) [102].

¹⁰ For further discussion regarding Microsoft's COM technologies, readers are referred to [36]

8.2.3.1 OMG OMA

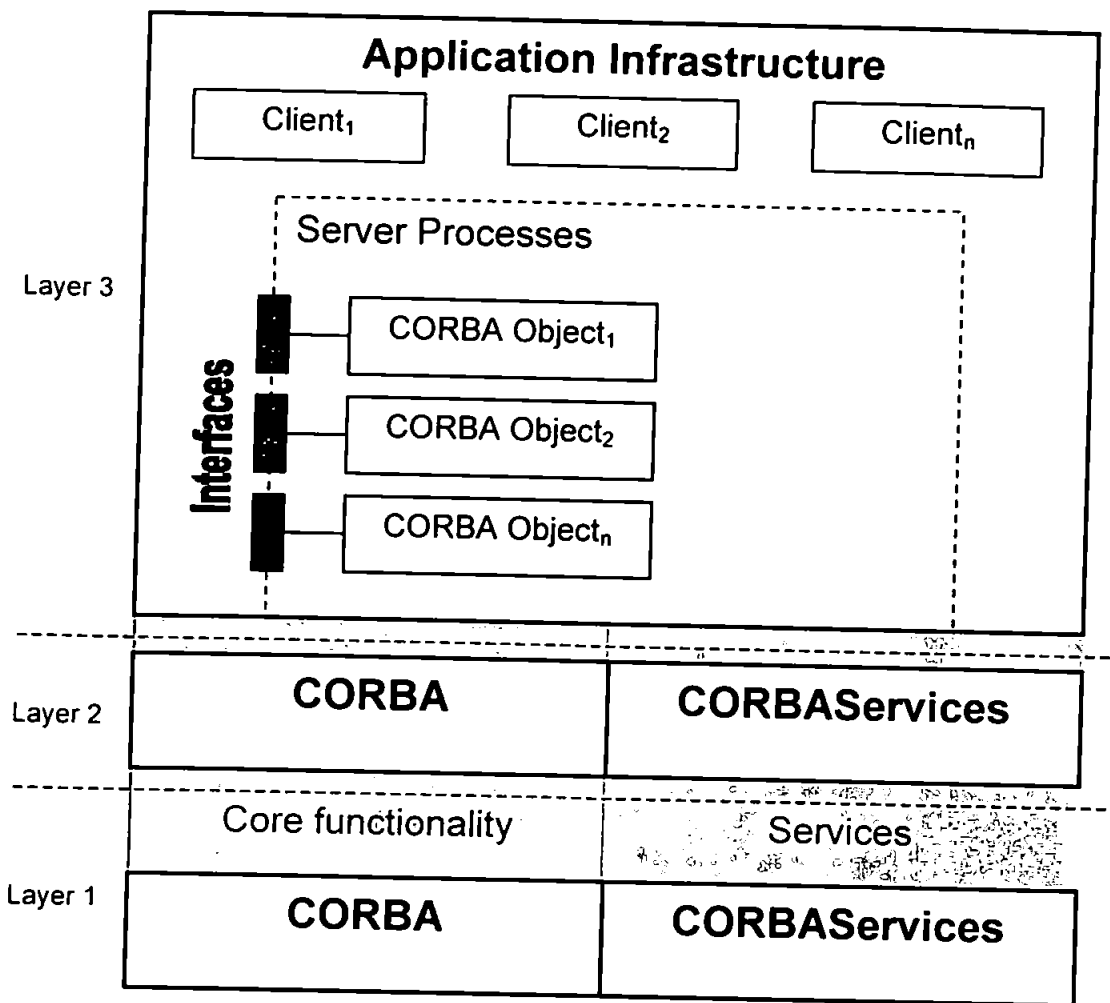


Figure 8-3 – Mapping the OMA to the Component Platform Reference Model

8.2.3.1.1 OMG OMA - Layer 1

Core distribution functionality is defined within the CORBA standard, which specifies both network protocols (running, in general, on top of TCP/IP) and marshalling. Distribution services are generally implemented as CORBAServices, which cover a large range of services, including security and naming.

8.2.3.1.2 *OMG OMA - Layer 2*

As the OMA specifically focuses upon distributed object solutions, layer 2 functionality is also provided within the CORBA specification. The structure of a CORBA 2 component is very much a contentious issue – it has been argued [132] that the CORBA 2 standard does not provide a component infrastructure. However, while a CORBA object may not have a formally defined component structure, the nature of CORBA objects (i.e. exposing functionality through interfaces, binary reusability) indicates a component-oriented nature. The issue of a component model for CORBA is to be addressed in the CORBA 3 specification. Additional functionality related to component infrastructure is provided via a set of system services (OMG Common Object Services, OMG Common Facilities). Further services related to component specific aspects, such as versioning, persistence and licensing can also be found within the CORBAServices specification.

8.2.3.1.3 *OMG OMA - Layer 3*

Components are implemented as CORBA objects. Common types of object for horizontal and vertical domains are defined in standards such as CORBAServices[109].

Interfaces for CORBA objects are defined using the OMG Interface Definition Language, Interfaces are then implemented by a CORBA object. Common interfaces within the OMA for industry specific implementations are provided within the CORBA domain initiatives [109].

For current CORBA implementations, component containers take the form of CORBA servers, which will create instances of CORBA objects and execute an event loop to deal with calls to these instances. True component containers and scripting techniques for the OMA are realised by

the CORBA Component Model and CORBA Component Scripting, which are integral parts of the CORBA 3.0 specification¹¹.

8.2.3.2 Microsoft Windows DNA

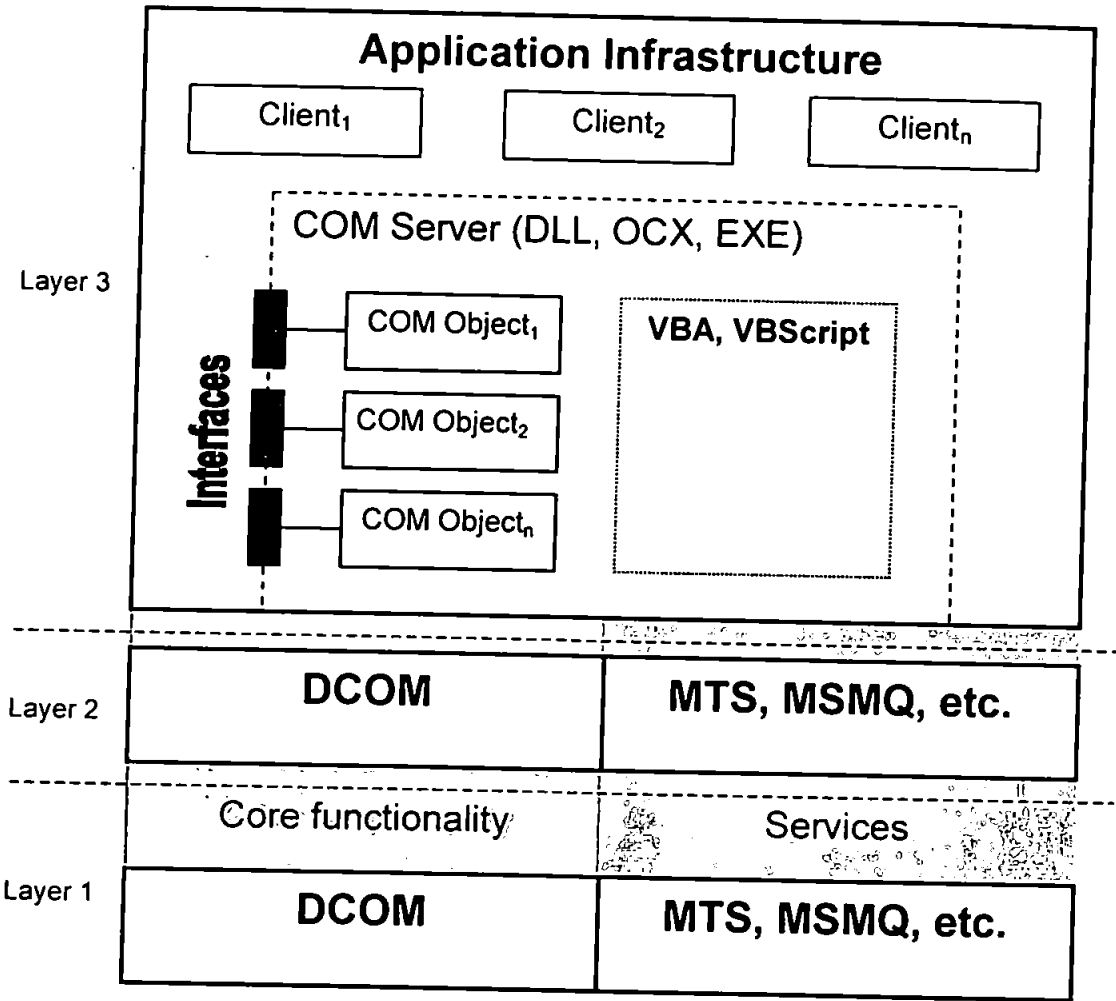


Figure 8-4 – Mapping the Windows DNA to the Component Platform Reference Model

¹¹ At the time of writing (September 2000), the CORBA 3 specification was not publicly available from the OMG.

8.2.3.2.1 Microsoft Windows DNA - Layer 1

The distribution infrastructure for the original DNA is provided by the DCOM standard implementation. As well as defining protocols, DCOM provides a number of distributed services, such as security, as part of the standard implementation.

8.2.3.2.2 Microsoft Windows DNA - Layer 2

DCOM also provides core component services for the DNA by specifying component structure and inter-component communication. Additional component services are generally provided via an implementation. For example, scaling, transaction control and further security are all provided by the Microsoft Transaction Server product, and asynchronous messaging is provided by the Microsoft Message Queue Server (MSMQ). However, the advent of COM+ [88] has affected the nature of the DNA, in particular at two lower layers of the reference model. While early versions of the DNA used DCOM to provide core functionality and various products to provide component services, the core COM+ implementation provides both core functionality and additional services (such as messaging, transactions, scaling and event monitoring) as part of a single implementation. The COM+ implementation can be seen as a complete layer 1 and 2 implementation.

8.2.3.2.3 Microsoft Windows DNA - Layer 3

Components within the DNA will be implemented as DCOM objects. As with the OMA, Microsoft have domain initiatives [to address common components for both horizontal and vertical domains.

Interfaces for components are defined using the Microsoft Interface Definition Language and are then mapped to component implementations. Unlike CORBA objects, a DNA component can

implement numerous interfaces. Various types of component within the DNA (for example, ActiveX controls, automation servers) are required to implement certain types of interfaces.

Clients within the DNA will generally be concerned with providing user interfaces to component functionality and are easily developed through scripting. While these can be developed as new applications using Windows development tools, standard clients such as Internet Explorer or the user interface elements of the Microsoft Office suite can also be used to access both standard and custom component functionality, by exploiting the scripting elements of the DNA.

Component containers come in many forms within the DNA, for example, ActiveX controls and COM DLLs. Containers can also exist in executable form, for example COM servers such as Excel or Word.

Finally, the scripting element of the DNA is provided by the Visual Basic for Applications (VBA) language, which is commonly used to script DCOM components.

Finally, it is interesting to note recent developments within the DNA. While the initial specification was very much focused upon DCOM and component products, the recent DNA 2000 release [102] has moved away from a pure component approach to mix components with other approaches, in particular Internet technologies such as XML, HTML and HTTP. This is similar in approach to the Netscient platform (see section 6.6.3.3), which attempted to use technologies to their strengths, rather than relying on a single technology for all aspects of development.

8.2.4 An Alternative Viewpoint – Visual Basic 3

In some respect, it is more interesting to relate the reference model to less well-known component platforms, or platforms that are not considered “pure” component approaches. One such example is an early version of the Visual Basic development environment (version 3) that provided a basic component platform for the development of Windows applications using 16 bit VBX components [91], especially once Microsoft had released the control development kit for third party developers. This is a far less “pure” component platform, but the reference model can still be used to assess aspects of the platform:

8.2.4.1 Visual Basic 3 Layer 1 – Distribution Infrastructure

No facilities for distributed applications were provided for the platform.

8.2.4.2 Visual Basic 3 Layer 2 – Component Infrastructure

A simple set of mechanisms that enabled the visual representation of a Visual Basic eXtension (VBX) control within the Visual Basic environment and an API for interfacing the environment and clients with control functionality.

- **Component structuring** – defined in the control development kit API to provide properties and events for a control, although no means of providing methods was included within the API.
- **Component services:** No additional services were defined as part of the platform.

8.2.4.3 Layer 3 – Application infrastructure

- **Components** – VBX controls were the core component type for the platform. They took the form of a 16 bit binary library that provided functionality that extends the Visual Basic environment in some way, generally additional GUI components.

- **Interfaces** – A control application programmer interface defined the structure of source code (event loop, common properties, extended properties, functions for reading and writing properties, functions for drawing the component on a form, etc.) that had to be used in order that the VBX containers could use (interface with) controls.
- **Clients** – Applications developed within the Visual Basic environment used VBX controls to build up the interface for a Windows application. These applications were then compiled into Windows executables.
- **Containers** – The core container for VBX controls was the Visual Basic development environment itself. However, while a single “component” could be provided via a VBX control, the control itself could also be considered a container for the component as it provided the execution environment for component functionality (which was very similar to a 16 bit Windows DLL structure). A client application required the VBX controls in its distribution in order to function.
- **Scripting** – The Visual Basic language itself, in its version 3 form, was little more than a basic scripting language – it providing core programming constructs but was loosely typed and had little memory management.

8.2.5 Applications of a Reference Model for Component Platforms

To conclude this section, the use of the reference model within the learning process is considered. As discussed at the beginning of this section, the classic use of reference models is the provision of a technology independent viewpoint of an aspect of the IT field. However, their importance as a learning tool should not be underestimated. In providing a technology independent view of, for example, a specific development approach, it demonstrates the decomposition of the topic into pure facets – i.e. concepts can be described independent of implementation. Our survey responses have demonstrated the problems within the field of component-orientation in marrying concepts

to implementation. If an organisation can clearly see the constituent parts of an approach, they are better able to relate it to their knowledge of previous techniques.

Thus, the developed reference model serves two purposes – firstly as a comparative tool but also, more importantly, as a learning tool to facilitate the relation of component-oriented concepts to both existing knowledge and the development of new understanding.

8.3 *Developing the Organisational Learning (OL) Perspective*

Of the approaches in chapter 3, organisational learning (OL) appears to be the most suitable for our aims as it focuses upon the development of organisational knowledge about an innovation. It differs from the other theories discussed in the chapter which are more focussed upon why a technology is adopted by an organisation, but not the learning process once the technology is adopted.

Following our review of OL in chapter 3, we have identified the following aspects that affect the development of results from our research:

1. Component-orientation is a complex technology that may present significant knowledge barriers for an organisation wishing to adopt it.
2. An organisation should have a knowledge base regarding a technology before attempting to adopt it.
3. Effective construction of a knowledge base requires information related to previous know-how/experience of a technology, rather than simple signalling information.
4. The knowledge base is built up from various levels of learning (individual, group and organisation) and places a need for the development of common language (specific to the adoption domain) throughout an organisation.

We therefore aim to provide information that can be used within the construction of an organisation's knowledge base regarding component-orientation and to address the issue of knowledge barriers for the organisation.

8.3.1 The Organisational Learning Process

An important difference between OL and previous adoption theories is the nature of both information and its communication. Attewell's [5] work in applying diffusion of innovations and OL theory to the adoption of complex technology is at the heart of the matter. It insists on the distinction between signalling information and knowledge. While this can be likened to the differentiation of hard and soft information within diffusion of innovations theory, the emphasis from the OL perspective is that of the communication and construction of knowledge. This differs from previous theories that focus upon signalling, viewing the communication of existence as the important aspect for the adoption of an innovation.

Attewell focuses upon the need for the communication of knowledge as the precursor to effective adoption of a technology, through the lowering of what he refers to as *knowledge barriers*. A knowledge barrier can be defined as an aspect of the new technology that the organisation needs to understand in order that it can be used effectively – the burden of developing such technical know-how becomes a hurdle to adoption. Such barriers are generally related to the elements of context as discussed in section 3.2.1.3. Attewell presents a number of potential solutions to overcoming knowledge barriers, and identifies the role of suppliers, or other external parties in the development of organisational knowledge. The advantage of the outside authority, he argues, lies in economies of scale with learning. An external consultant or supplier is likely to have a great deal more experience in the use of the new technology, and they have had greater potential to use it. Another benefit Attewell identifies from use of an external source is "rare event learning" – obtaining knowledge from events that occur infrequently. He argues that there is great

value in such experience, as such exceptional experiences generally occur when new approaches are being used – the adopter can learn from the consultant's "mistakes".

However, while the supplier or consultant plays a role in lowering the knowledge barriers in an organisation, it is also acknowledged that organisational knowledge development has to come from "learning by doing" for an organisation. Attewell argues that with complex technology, its effect upon practice, users and products is unique to an organisation and it is only through using a new technology that a comprehensive knowledge base can be created.

Therefore, we can view our development of results not as providing a complete knowledge base that is transferable directly into an organisation, but to guide an organisation's own development. Any development should aid in the identification of potentially problematic areas and encourage learning from the experience of others – a coaching, rather than dictatorial role in the development of organisational knowledge.

However, while Attewell's work guides in setting the direction for the development of results, it makes no suggestions as to how this knowledge is communicated or the process of assimilation. For these aspects, other work provides guidance.

Fichman & Kemerer [54] use Attewell's work as a focus for their own research into the adoption of object oriented technologies. However, they develop the background theory by defining the assimilation process – the process by which knowledge regarding a technology is developed within an organisation. They define the process as:

1. Grasping abstract principles of the technology.
2. Understanding the nature of benefits attributable to the technology.

3. Grasping specific technical features of different commercially available instances of the technology.
4. Discerning the kinds of problems to which the technology is best applied.
5. Acquiring individual skills and knowledge needed to produce a sound technical product on particular development projects.
6. Designing appropriate organisational changes in terms of the team structure, hiring, training and incentives.

We can focus on these activities in presenting the research results, and complement them by providing tools to aid in organisational development.

Crossan, Lane & White [43] identify the different levels at which learning occurs within the organisation (individual, group and organisation) and how these are interrelated. This harks back to Attewell's work, which comments upon the need for individual learning to be the foundation for organisational knowledge. In both instances, organisation learning results from the development of individual knowledge into organisational culture – the *institutionalisation* of the 4Is framework.

In relating the 4Is framework to the aims of the research programme, focus should be upon the interpreting and integrating processes. It is at this level that learning moves from being an individual activity to a group level with shared understanding and the development of organisational knowledge.

Within both of these processes, the authors stress the importance of language within the development of knowledge. A shared, common language enables conversation about the technology that, in turn, supports the learning process. As the authors state:

"conversation can be used not only to convey meaning, but also to evolve new meaning."

The value of conversation is also backed up by other complementary learning theories, such as social constructivism [61], that identify the value of social interaction among peers in the learner process. It is also featured in survey responses (see section 7.4.3), that comment upon a need for common language among all that are involved in the learning process within an organisation.

When considering the role of language in the learning of a new technology, we focus upon the commonality of definition and concepts – to ensure that everyone involved in the learning process has a shared understanding of the technology. Within the DOLMEN case study a lack of shared understanding hampered development efforts, and a common issue drawn from the practitioner survey was that of reconciling concepts with implementation. Additionally, such an issue is also important in Fichman & Kemerer's assimilation process, as they define the grasping of abstract principles as the first activity in the process. Common language can be seen as a way to address this issue.

To summarise, the OL approach provides a number of ideas that are germane to the development of research results:

1. The external information provider can play a pivotal role in the development of knowledge regarding a complex technology.
2. Rare event learning has value in the development of knowledge.
3. Learning by doing should be guided from external experience.

4. Interpretation and integration are supported through the use of language

8.4 Approaches to Adoption

The following examines a number of approaches to the adoption of new technologies and techniques in the light of the criteria identified above.

8.4.1 Standards/Guidelines

There are many standards and guidelines within in the field of software engineering [159].

Standards are described by ISO [79] as:

...documented agreements containing technical specifications or other precise criteria to be used consistently as rules, guidelines, or definitions to characteristics, to ensure that materials, products, processes and services are fit for their purpose.

The difference between standards and guidelines within software engineering is somewhat blurred. However, Sommerville [145] distinguishes them in terms of rigour. A standard will be very prescriptive defining a specific approach that must be used in achieving an outcome, whereas guidelines will generally be more advisory in nature. However, in both cases they will generally go through a very thorough committee based submission, assessment and voting process before being defined and published. For example, both the ISO and the OMG have a well-defined process for the development of their standards involving proposal, submission, review and voting procedures.

While a standard could be considered as part of the institutionalisation of a learning organisation, in that it defines common vocabulary, language and practice in achieving certain aims, we cannot consider them a suitable approach for our own needs in the development of our results. Firstly,

our results are not in a suitable form to be cast as contributions toward guidelines or standards for the adoption and use of component technologies. Our case studies have allowed us to develop theories in the adoption and use of component technologies, and the survey has allowed us to focus these theories. However we cannot, and do not want to, state that the occurrence from the case studies are indicative of practice for others using component technologies. Additionally, we do not wish to dictate practice from these results – we wish to communicate our own experiences so that others can reuse them. Therefore, we cannot consider a standards/guidelines approach as suitable for our aims in developing our results.

8.4.2 Transfer Packages

Transfer packages take the form of documentation related to a specific software technology or technique in order to guide adopters. These packages will generally package data regarding the technology or technique either from experimentation or review, and are intended to guide future use. Two types of package are considered below, the Transition Package from the Software Engineering Institute (SEI) and the Software Engineering Laboratory's Experience Package concept.

8.4.2.1 SEI Transition Packages

The SEI Transition Package (TxP) was developed within the SEI's Transition Enabling Program [140]. Fowler and Patrick detail the concepts behind transition packages, provide an example of such a package (in this case, for requirements management, a key process area within the CMM [116]) and detail the process of their creation in [57].

The authors describe a transition package as:

"a kit based approach to providing materials needed to use new technologies and practices as well as to introduce technologies and practices into organizations" (pp. 1).

They define the following as the process for the production of a transition package:

1. Document a description of both the subject area for the transition package and the people you expect to use it. This description establishes the scope and purpose of your transition package effort.
2. Identify potential sources of materials.
3. Gather the materials.
4. Identify multiple views of the materials; if possible, base views on accepted reference models.
5. Assemble and package the materials, and create the views.
6. Distribute the package to the users.
7. Evaluate how people use the TxP and upgrade it accordingly.

They also comment that feedback from their evaluation of transition packages has highlighted user interest in evidence being presented in a case study format – having a context in which to understand the material. This is clearly very relevant to the presentation of the results from this research programme.

The transition package concept is interesting in considering approaches to the adoption using diffusion of innovations theory as its foundation. Transition packages draw from diffusion of innovations and aim packages at early majority to late adopter types. As such, it seems that a lot of information presented within the package is signalling information, rather than knowledge. While a lot is discussed relating to multiple views and sources of evidence, little consideration is given to the provision of knowledge or experience related to the actual use of the technology. For

this reason, we consider that transition packages do not provide a completely satisfactory technique for our purposes.

8.4.2.2 SEL Experience Packages

The SEL approach differs from transition packages in that it exists within a process improvement approach (the Quality Improvement Paradigm), and is based very much upon experience and experimentation. The Experience Factory approach [11] views process improvement as an iterative process of understanding, assessing and packaging. As part of this approach the experience package presents assessment results in the form of tools, standards and training materials that will aid in the improvement of the development process with respect to a specific aspect of software development. Experience packages within the SEL's own experience factory [13] have included an Ada users manual, a cleanroom process model and a software management environment.

The process of developing the packages forms part of the Quality Improvement Paradigm, with measurement arising from the Goal/Question/Metric paradigm. The approach is discussed in section 3.1.4.1. The packages then form part of an *experience base*, that the organisation associated with the experience factory can draw from when developing their software processes.

Experience packages have their foundation in quantitative data obtained from controlled measurement within the practitioner environment. This is a contrast to our own data collection methods that focused upon case study of external development projects. The experience factory is is, therefore, not suitable for the development of packages from the case studies as no measurements were defined for each project. As stated in the case study discussion, there was no management role in either project – the researcher could not dictate project direction. This problem was also addressed by work by DaimlerChrysler AG in trying to establish their own

experience factory. This work, initially documented by Houdek [72], identified several problems in using the approach within the DaimlerChrysler organisation – namely insufficient structure, unsuitable classification and missing technical support. Landes, Schnieder and Houdek took up these issues within the OL context in a later publication [93], where the problem of lack of quantitative data was discussed explicitly. The information gained from experience in their own projects were primarily reports detailing problems with a technology or technique, explanations for the problems, attempts at solutions and solution evaluation. However, rather than attempt to restructure application projects within the organisation the authors examined what could be done with the type of information they had at their disposal. In an aim to provide structure around qualitative information related to experiences, they developed the concept of *quality patterns* – drawing from the wider concept of patterns [2] that is discussed below.

In considering the work from DaimlerChrysler, we can see a number of similarities with our own research – it is grounded in OL theory and it involves the development of qualitative evidence into representative experience. The major difference is that internal experience was used for the DaimlerChrysler work, whereas we aim to develop our results as an external input into the OL process. However, as discussed above, the external input to the process can often be beneficial to the organisation. The following section considers the pattern approach in more detail, before assessing its suitability against our own criteria.

8.5 Pattern Approaches

A pattern can be defined as a problem/solution pair – it defines a problem and puts forward suggestions for a solution, both the problem and solution being based upon experience. The widely cited origin of patterns comes from Christopher Alexander, who observed this problem/solution pairing as the way experts approach problem solving and applied the theory to architecture [2]. Alexander observed that experts would never consider a completely new solution

for a problem, they would base a new solution upon previous solutions – the reuse of experience in a problem solving capacity.

The main feature of patterns is that they are problem-, rather than solution-oriented. This approach is quite different from standards, which prescribe a way of practice in order to improve the quality of, for example, software development products. Houdek & Kempter [71] summarise the pattern approach effectively as:

In describing a pattern, the main focus is not only in presentation a solution, but in observing what problems a user of this experience will have in the future. The solution is described with respect to future use.

The most well known use of patterns within the software engineering field is the use of design patterns for object oriented systems [60]. This now seminal work applies the patterns concept to the design of object orientated systems, defining a group of patterns that address problems in the design of object oriented systems and putting forward clear, simple solutions.

However, while patterns have typically been applied to design problems due, perhaps, to their origins, there are becoming more diverse, with applications occurring wherever a problem oriented approach seems appropriate (for example, in architecture [2] and organisational development [15]). An early example of the use of patterns for education comes from Cunningham & Beck [44], who developed a small pattern language for novice developers wishing to learn about Smalltalk. While these patterns focused upon implementation issues (for example the development of a windows based GUI), they do demonstrate the possibility of using patterns for learning.

Another interesting development is the concept of AntiPatterns. AntiPatterns are described by Brown et. al. [29] as a pattern to describe a problem area within (in this case) software

development. While a pattern aims to guide best practice, an AntiPattern aims to identify a problem and describe how to obtain a good solution from the situation.

A pattern approach generally consists of the following:

- **Pattern language:** A collection of interrelated patterns is referred to as a pattern language. While it does not enforce a formal language, it encourages a common vocabulary for talking about the particular domain or problem.
- **Pattern template:** Defining a structure for all patterns within the pattern collection, also known as a *pattern catalog*.
- **Patterns:** A pattern will address an individual problem within a domain, present possible solutions, and relate them to others.

Another common element of patterns is their complementary nature. Generally, a pattern will not exist in isolation, but will complement other patterns, either on a peer level (termed synergistic by Mowbray and Malveau [105]) or by contributing to a larger scale pattern (termed subsidiary patterns by Mowbray and Malveau).

8.5.1 Examples of Patterns

8.5.1.1 Alexander's Architectural Patterns

- **Pattern language:** "Towns, Buildings and Construction" was the term given to Alexander et. al.'s pattern language [3], related to the architecture within towns. The language covers a huge range (253 patterns) of concepts related to town design, ranging from the large, town scale (for example, the Independent Region) to the small, related to a single building (for example, Alcove).

- **Template:** Alexander's patterns have little formal structure. Each does, however, follow a general form (sometimes referred to as Alexanderian Form [29]). Firstly, the pattern is assigned a name, followed by an overview of the pattern and the problem it addresses. A description of the pattern and a number of examples follow this. This description will be followed by a "therefore", which will detail a solution to the problem identified in the pattern, along with detail of other patterns that complement the solution.
- **Pattern example:** A simple example from the pattern language is a Window Place. This pattern relates to a location within a room that allows the occupant can sit and also have a good source of light. Examples such as a window seat, a bay window and a low sill. The pattern's solution is to provide a Window Place in any room where the occupant will spend a length of time during the day, and the solution is related to other patterns such as Alcoves, Low Sill and Built-in Seats.

8.5.1.2 Gamma et. al.'s Design Patterns for Object-orientation Systems [60]

- **Pattern language:** The pattern language of Gamma et. al.'s patterns relates to the design of object-orientation systems, expressing solutions in terms of classes and objects that work together to address a problem.
- **Template:** The authors have a general structure for the pattern that is followed by most patterns related to software development. There is a name, a problem that describes when the pattern should be applied, a solution that describes the elements that address the problem, and consequences, that are the results and trade-offs coming from using the pattern. However, each pattern is also defined in greater detail using a template:

Name and classification: The pattern name and how it relates to the pattern language.

The authors define a number of classifications based upon purpose (creational, structural or behavioural) and scope (class or object).

Intent: What is the pattern's intention

Also known as: Other names for the pattern

Motivation: The “problem/solution” aspect of the pattern – an instance that illustrates the design problem and how the classes and objects within the pattern addressed it.

Applicability: Where the pattern can be applied

Structure: A diagrammatic representation of the objects and classes within the pattern, defined in OMT [135].

Participants: A description of the classes and objects within the pattern

Collaborations: How the participants collaborate

Consequences: The trade-offs and results of using the pattern

Implementation: Advice on how to implement the pattern

Sample Code: An illustration of the pattern in an OO language

Known Uses: Examples of the pattern in real world systems

Related Patterns: Other patterns that complement the pattern, on a synergistic or subsidiary level.

- **Pattern example:** A widely known and used pattern from the OO design language is the Observer pattern. This pattern identifies the need for other objects to be informed of the change in state of a given object. Its motivation comes from the model/view paradigm [97] inherent in windows systems, where different views of a specific data set are provided. These views need to be updated if the data changes. Hence, the views are observers on the data object. The pattern defines the pattern participants (e.g. subject, observer) and how they relate. It goes on to discuss a number of implementation issues, such as mapping subjects to observer, dealing with more than one observer and how the update is triggered.

8.5.1.3 Quality Patterns

- **Pattern language:** Quality patterns form the basis for the communication of experience within an Experience Factory context. Therefore, the author’s quality patterns can be seen to

be an experience package, used by others within the organisation to learn. The language defined by such patterns relates to the communication of experience with specific development processes and technologies.

- **Template:** The template follows a similar general form of other pattern approaches in that it provides a pattern name, a problem, and solution and an explanation. The problem/solution pairing within the pattern can be considered the experience aspect of the pattern, which holds the learned knowledge. The context in which the knowledge is placed enables the transference of the experience. The general pattern structure is expanded to a formal pattern template consisting of:

Classification: Broken into package and object types, and a viewpoint, relating to the typical user of the pattern.

Abstract: An overview of the pattern

Problem: Defining the source of the pattern

Solution: A model solution to the problem presented

Context: Where such a pattern would be relevant

Example: Describing a use of the pattern in a given situation

Explanation: A description of the use of the pattern and its outcomes

Related experience: Relationships with other patterns within the experience package.

Administrative information: Author name, date pattern produced, etc.

- **Pattern example:** An example provided by the authors in [71] is that of an IT contract. The quality pattern examines the issues to address in the reviewing and amending of contracts, in particular related to the issue of outsourcing. It defines a number of issues to check in the provisioning of such contracts and places the issues in the context of large-scale development projects where development is outsourced.

8.5.2 Consideration of Patterns from an OL Perspective

Schmidt, Johnson & Fayed [136] identified the following as motivation for the creation of patterns and pattern languages:

1. Success is more important than novelty: A pattern becomes more valuable the longer it has been used successfully.
2. Emphasis on writing and clarity of communication: Through the use of a template, patterns can follow a common form for ease of communication.
3. Qualitative evaluation of knowledge: Knowledge about problems can be expressed in a qualitative way, rather than in a quantitative way or through theorising.
4. Good patterns arise from practical experience
5. Recognise the important of human dimensions in software development: Patterns aim to support the human nature of software development, rather than trying to enforce rigid rules, or replace the human element with automated tools.

For our own needs and from the theoretical viewpoint we have developed from literature review (i.e. practitioner focused research identifying issues that should be shared with others in the field), points 2-5 all provide a good argument for a patterns based approach. We have identified communication as an essential aspect of the learning experience, we focus upon qualitative data from our research, we wish to draw from and share practical experience, and we recognise the human aspect of the learning process.

We also see two other issues as important in considering the use of patterns from an OL context. Firstly, they communicate expert knowledge and experience – they aim to help the user by sharing previous experience. The pattern can be used as an external source, as discussed by Attewell to aid in the learning process through the lowering of knowledge barriers. Attewell's

identification of the communication of know-how or knowledge as the essence of organisational learning can also be addressed with such an approach, as can the value of rare-event learning.

More importantly, patterns specifically aim to define a language that is used when discussing issues within the domain. The influence of OO design patterns upon the field of object-orientation can be seen in a language such as Java, which defines, for example, Observer classes, and in component approaches. The observer pattern is very much an influence upon component monitoring such as that defined in the CORBA Event Service [110] and also interception in COM+ [88]. As previously emphasised in the discussion regarding organisational learning, language is extremely important in the development of individual learning into an organisational context.

8.6 Conclusions: An Overall Strategy for Results Development

In concluding this investigation into a strategy for the development of research results, all of the ideas discussed within this chapter are drawn together. A reference model for component platforms was developed, and its use as a learning tool was discussed. Previous approaches to the transition of experience were examined and a pattern-based approach was identified as being the most suitable solution to our needs. However, the findings from other approaches should not be dismissed, in particular the experiences of the SEI [57] in the validation of their transfer package, which highlighted the need for context in such things. Therefore, the overall strategy focuses upon a pattern language for the adoption and use of component-orientation, but places the language in the context from which it is developed. The complete package is presented in the following chapter. However, following identification of an approach for the package based upon literature and the type of results we had obtained from study, an initial package was developed. This package was then reviewed by an industrial software development organisation in order to further refine the approach used. This review is discussed below.

8.6.1 Refinement Based upon Industrial Feedback

As the intention of the package is that it should be used to promote knowledge regarding component-orientation in industry, it was important to obtain industrial input into the development of the package. A draft package was produced and assessed by a research and development group within a large software/telecommunications organisation in Germany. The group specialises in the development of software solutions using leading edge technologies, in order to determine their effectiveness for other product lines. As one of the technologies they were currently hoping to use was component-orientation, there was a good opportunity to examine the suitability of approach to the development of knowledge in this area. The package was delivered to the organisation and distributed among lead developers for use in considering the suitability of component-orientation to specific development projects. Feedback in the suitability of the package approach was very positive. Particular issues drawn from the feedback were:

1. The patterns approach is good – the problem/solution pairing, backed up with examples of real world occurrences, is a very good format for presenting experience. The fact that this knowledge comes from real world experience differs from a lot of literature about CBSD, which seems to dictate practice without demonstrating any foundation for the arguments presented.
2. A non-prescriptive format is also effective – it is very difficult to get developers working to tight deadlines to follow approaches that enforce specific practice. The “softer” approach provided by patterns enables their use without dictating practice.

3. The context is very valuable – its both demonstrates the origins of the knowledge and also enables the user to be able to relate the suitability of the package to the user's own needs. The reference model clearly defines aspects of component-orientation free from implementation – again, the majority of literature tends to express choice of platform from a very vendor-specific view.

The following, from an email discussion with one developer, highlights the perception of the package in the organisation:

> Do you see value in the package for educating in the use of CBSD?

Yes, the package can be viewed as some sort of "best practice" which supports developing CBSD. From one of the projects I got in touch with here at XXX, they say that if they would have used your package which says "First, discover the technology, etc.", they wouldn't have blown a lot of money!

However, it was important to obtain feedback that would help refine the package into a more effective tool¹². In using the feedback for refinement, a couple of issues were identified:

1. The focus of the package should be the patterns – the context complements the patterns well but should not have as much emphasis in the package. A brief description of each case study (type of industry, scale, use of technologies) is sufficient.
2. Greater emphasis should be made regarding the reference model – it is useful as a learning tool in its own right, as well as being used for comparison of case study platforms.

¹² The draft package discussed the case studies in detail as it was considered important to provide a complete picture of each. Additionally, the reference model was only used for comparison of case study approaches, it was presented as a subsection of the case study discussion.

Following the feedback, the package was refined in the following ways:

1. Case study descriptions were reduced to brief reviews of important aspects. A lot of descriptive text was removed.
2. The reference model was moved into a section on its own, prior to the discussion of its use within the case studies.
3. Further description was added to the patterns, in particular the example of occurrences of the patterns were strengthened.

The following details the structure of the revised package, based upon its refinement following industrial feedback.

8.6.2 Package Structure

8.6.2.1 Context

Detailing each situation that has contributed to the development of the pattern language. The context is composed of the following aspects:

- **Reference model for component platforms:** Used to define core concepts in component-orientation and also as a comparative tool in the case studies.
- **Points of reference from the case studies:** Each case study is defined from three points of reference:
 - **Overview:** A textual description of the case study, detailing type of industry, scale of project, type of software developed, etc.
 - **Component platform:** Based upon the reference model for component platforms.

- **Development process:** The nature of the process used to develop case study software products.
- **Survey element:** Additionally, the survey aspect of the research is placed in the context of strengthening theories and focusing development. The survey aims and respondent profile are provided.

8.6.2.2 Language

The definition of the language itself, consists of:

- **Pattern template:** Drawing from templates within the field, the pattern template defines context, problem, solution and relationship within each pattern.
- **Patterns for the adoption and use of component technologies:** The patterns themselves, based on problems/solutions from the case studies, focussed by survey findings.

8.7 Summary

In considering the development of results from the research programme, knowledge barriers to the adoption of component technologies that relate to the complexity of learning have been identified. In aiming to overcome this barrier, the organisational learning field is further examined to consider previous attempts at technology adoption based upon this sound theoretical foundation. Approaches used to package experience related to process and product technologies were also considered and a pattern approach was identified as being the most suitable. However, the importance of context for the developed results is acknowledged. The reference model for component platforms is defined for two reasons. Firstly, as with other reference models within the software-engineering field, it can be used to compare different platforms to distinguish features. Additionally, it is used as a learning tool to differentiate concepts of component development from the complexity of technologies and implementations within the field. Once there is familiarity with concepts, the mapping of platforms to the reference model can help demonstrate

where each technology fits into the make up of a component system. A general strategy has been defined, and a draft implementation was tested in an industrial context. This enabled the development of the package into a full solution, detailed in the following chapter.

This chapter develops a transition package for use by organisations wishing to adopt and use component technologies. It can be seen as a culmination of literature review, data analysis and results development.

9. A Strategy for the Sharing of Experience in the Adoption and Use of Component Technologies

This chapter draws together results from the case studies and the practitioner survey, and literature related to the adoption of complex technologies to present a transition package that aims to assist organisations wishing to adopt and use component technology. Based upon theories of organisational learning, the package centres around a pattern language as a way of relating previous knowledge related to the use of component technologies. The aim is to provide a non-prescriptive approach to the use of component technologies, illustrating past experience to promote awareness, without explicitly dictating practice. The pattern language's source is presented as a context to the language, with a view to helping adopters relate the language to their own needs.

The majority of this chapter consists of the transition package, divided into context for the patterns and the pattern language itself. The package appears in the thesis as it would be provided to practitioners, although in order to avoid duplication, some aspects are referenced to other sections. The chapter then reviews the structure of the package and considers its use. Conclusions are drawn from this validation regarding the future development of such a package.

9.1 A Transition Package for the Adoption and Use of Component Technologies

The package consists of two sections: context, which discusses the nature of the work that led to the experiences expressed in the patterns, and the patterns themselves.

9.1.1 Target Audience

The package is intended for use by personnel who will be involved in the development of software using component-based techniques. The three primary roles to which the language is applied are aligned with viewpoints defined in the AntiPatterns reference model [29]:

1. Project managers
2. Software designers/architects
3. Software developers

Patterns are marked as being applicable to specific roles. It should be noted that these roles are deliberately broad - the package aims to present the learner with information and a context for that information. From this position, the learner can decide how relevant a specific pattern is to their own needs. The role markings are therefore for guidance only.

9.2 Context

The context section of the package describes the source of the pattern language, which is based upon practitioner experience in the use of component technologies. The context comprises the following sections:

- **A Reference Model for Component Platforms:** This reference model defines an implementation independent view of a component platform – the collection of software technologies used for the development of software systems. It enables the comparison of platforms from the different case studies (see below), and defines common elements of component-based systems in a generic way.
- **Case Study Points of Reference:** Depth of information related to the use of component technologies comes from case studies of component-orientation in practice. Two case

studies examined the use of component technologies in different settings. Outcomes from the case studies were in the form of theories related to the effect of component technologies upon software development.

- **Survey Points of Reference:** Case study outcomes provided a number of theories related to the effect of component technologies upon software development. However, in some cases it was difficult to determine whether outcomes were as a direct result of component technology, or whether a combination of factors was to blame. A survey of component practitioners enabled further clarification of issues, identifying common problems and isolating phenomena.

9.2.1 Reference Model for Component Platforms

The reference model discussed in section 8.2 is used to contrast the different development technologies used in each case study. As such, it is included as an aspect of this package. The definition of the reference model, from section 8.2.2, is included in the package as a technology independent view of a component platform that defines core concept.

9.2.2 Case Study Points of Reference

Each case study is defined from three points of reference:

- **Overview:** A textual description of the case study, detailing type of industry, scale of project, type of software developed, use of component technology.
- **Component platform:** Based upon the reference model for component platforms, in order to detail the software technologies used to meet project aims.
- **Development process:** The nature of the process used to develop products.

9.2.2.1 Case Study A - Points of Reference

9.2.2.1.1 Overview

Case Study A was a component-based project related to the development of a telecommunications architecture across disparate network technologies. It was particularly focussed upon the integration of mobile and fixed network technologies. It was a project whose development teams were distributed across Europe, with approximately thirty developers in eight different locations. Architectural designers were also distributed in other locations across Europe.

The development effort in the project centred on three aspects:

1. an integrated CORBA platform across mobile and fixed networks - to enable the interoperation of developed components independent of underlying technologies;
2. a component suite encapsulating the functionality of the defined telecommunications architecture;
3. application development that would make use of the component suite and component platform and test out the functionality of the telecommunications architecture.

9.2.2.1.2 Case Study A – Component Platform

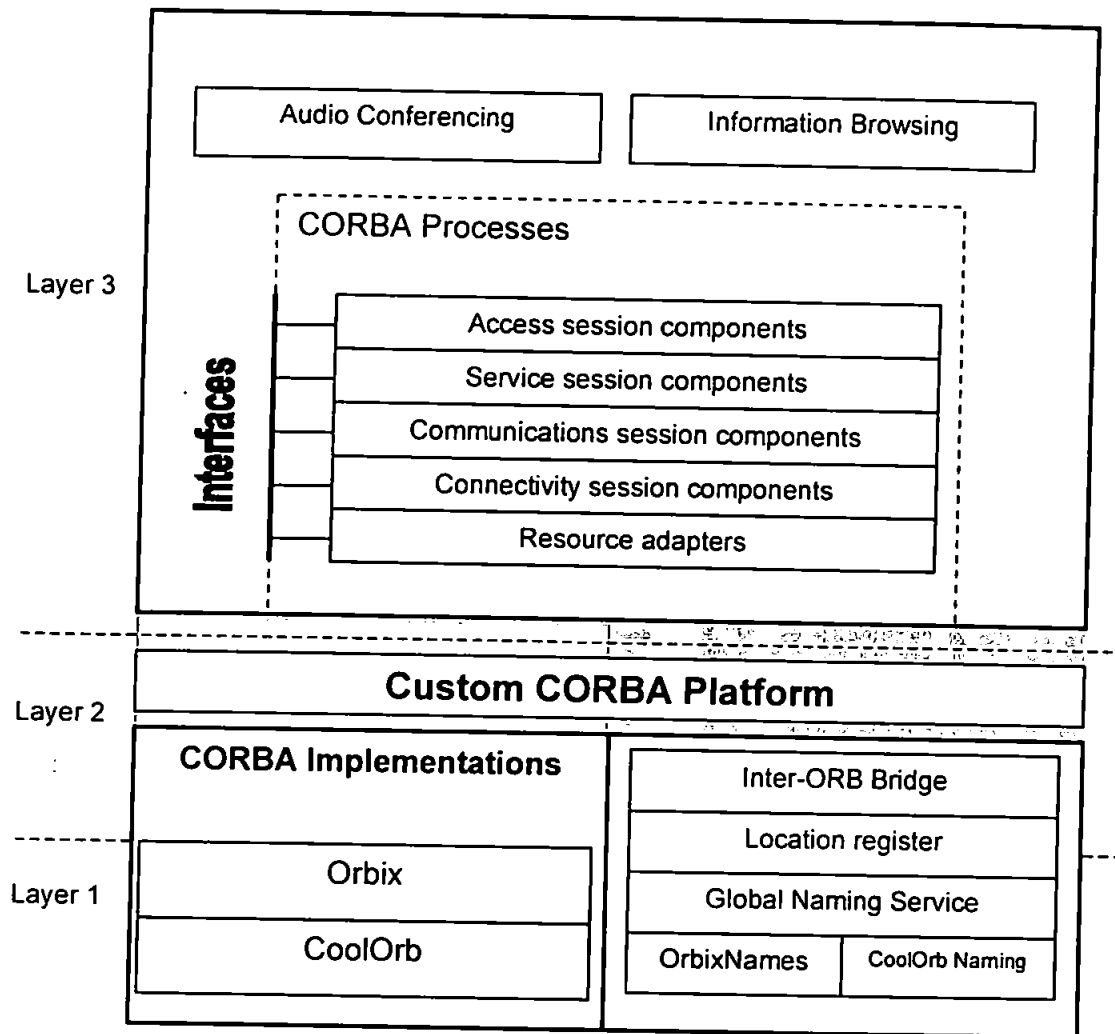


Figure 9-1 The component platform used in Case Study A

Notes on the component platform:

- The platform merges distribution and component functionality as all technologies were distributed in nature.
- GNS refers to Global Naming Service – a project-developed service to enable components from different CORBA implementations to be accessed independent of ORB.
- Location register was a service developed by the project to locate components on mobile platforms

- Component layers performed different aspects of functionality within the architecture.
Component interfaces are defined as OMG IDL.

9.2.2.1.3 Case Study A – Development Process

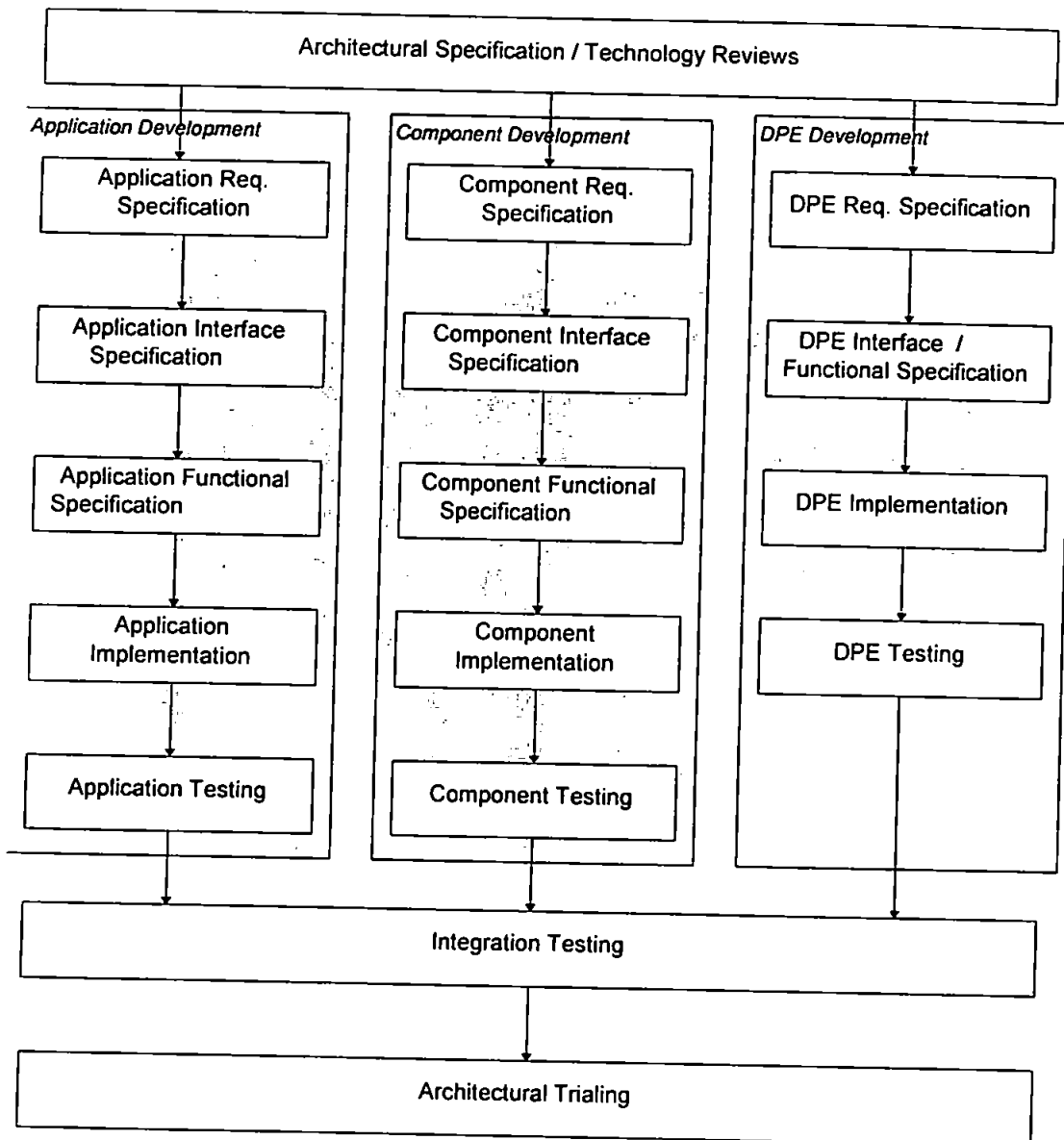


Figure 9-2 –The development process used in Case Study A

Notes on the development process:

- DPE refers to Distributed Processing Environment, the custom CORBA platform developed by the project to enable interoperation across core platforms.

- Technical reviews were literature-based assessments of the development technologies available to aid in the development of the software architecture.

9.2.2.2 Case Study B – Points of Reference

9.2.2.2.1 Case Study B - Overview

Case Study B centred on a network management Independent Software Vendor (ISV) in their first year of operation. The organisation was an SME, with the three directors and approximately ten software developers based in a central location. However, the organisation also relied on the services of external contractors and consultants for skills outside of their core domain. Its software development effort centred around two aspects:

1. The development of a software product line.
2. The development of in house software to support software product line development.

Two development teams were present within the organisation – six developers working on product development and four developers working on in house software development. The lead developer from each team also acted in the role of designer for the relevant software. The three company directors also provided input into requirements analysis for the organisation and had design input into both software projects.

9.2.2.2.2 Case Study B – Component Platform

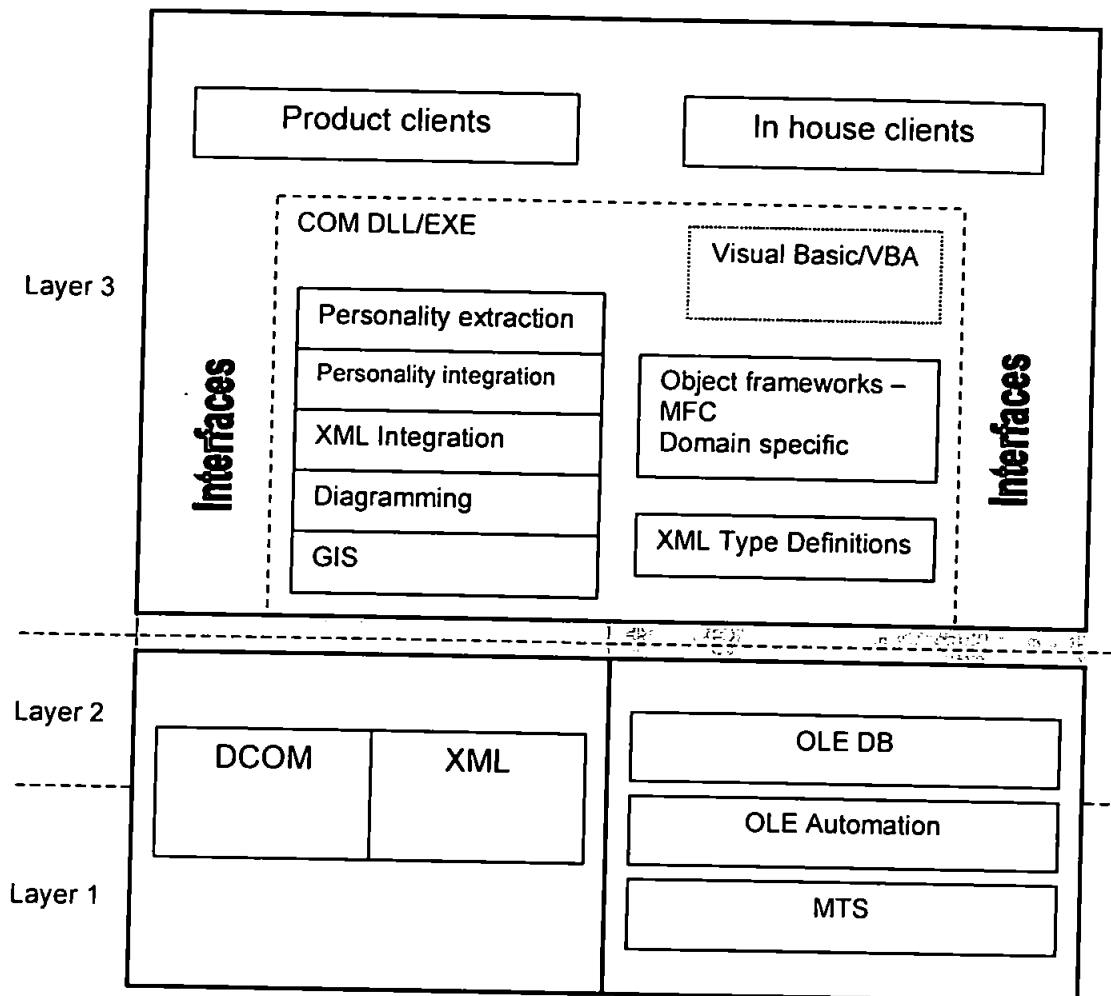


Figure 9-3 - The software platform used in Case Study B

Notes on the software platform:

- As with Case Study A, distribution was implicit in the software platform, hence no distinction is made between component and distribution functionality.
- Case Study B mixed other software techniques with component technologies (see Mixed Platforms pattern).
- Interfaces to components were defined in VB class definitions and Microsoft IDL.

- Other interfaces are provided by XML type definitions (see *Information Interface* pattern) and object definitions

9.2.2.2.3 Case Study B – Development Process

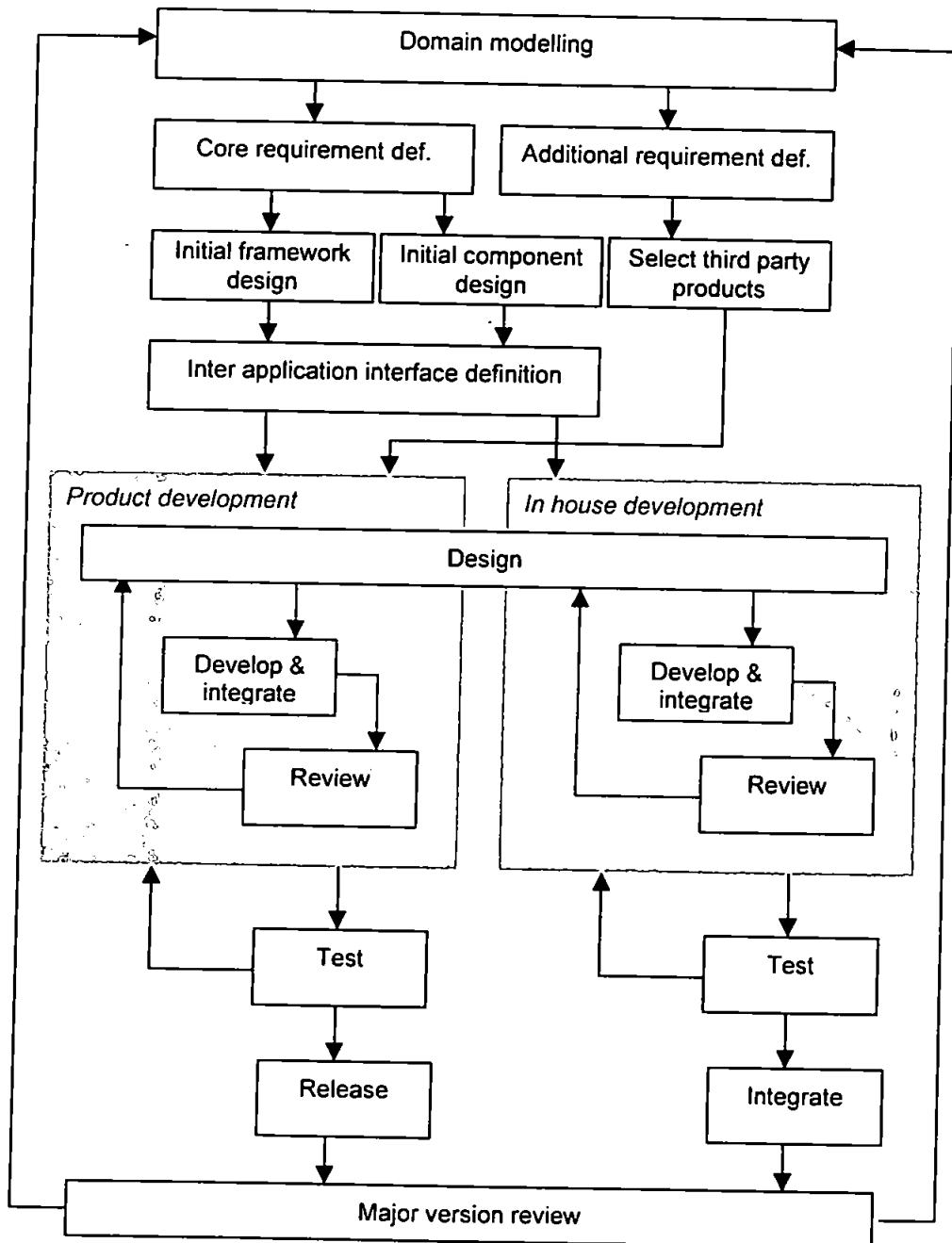


Figure 9-4 –The development process used in Case Study B

9.2.3 Survey Points of Reference

9.2.3.1 Survey Overview

The surveying of practitioners enabled a refinement of issues identified from case studies. The questionnaire was constructed to elicit opinion about findings from the case studies. The main conclusion that could be drawn from survey responses was that the most complex issue in the adoption and use of component technologies lies in the learning process. This finding situated transition information toward the development of knowledge from experience.

The target audience for the survey comprised 200 practitioners with experience of component-based technologies. Email addresses for potential respondents were collected from mailing lists that related to issues in component based development. Response rate for the survey was 22%, with 43 respondents.

9.2.3.2 Survey Questionnaire

Provided in appendix D

9.2.3.3 Respondents Details

Provided in appendix E.

9.3 Language

The vehicle used to communicate experience in the adoption and use of component technologies is a set of patterns that draw knowledge related to component-orientation from the case studies discussed above. The patterns take the standard form of problem/solution pairs based upon practical experience. They aim to identify potential problems in the use of component technologies, provide examples of these and suggest potential solutions.

9.3.1 Pattern template

The template follows a structured approach but is not as categorised as some. It aims to order experience, relying on qualitative description within the problem/example/solution aspects of each pattern for the core discussion and includes:

- **Name:** The name of the pattern.
- **Applies to:** Identification of personnel that are affected by this pattern (see section 9.1.1).
The case studies have demonstrated that component technologies have an effect upon most aspects of software development.
- **Abstract:** An overview of the pattern
- **Problem:** The origin of the pattern, and the problems it addresses
- **Example/experience:** A demonstration of the pattern drawing from case studies and survey responses.
- **Solution:** A possible solution to the problem, drawn from experience or “lessons learned” from the case studies.
- **Related to:** Other patterns within the language that have a relationship with the pattern.

9.3.2 Pattern Relationships

Relationships among patterns develop underlying concepts in component-orientation. Some patterns may have synergistic relationships, sharing a common principle or contributory outcome. Other patterns may complement others through the affect of outcomes resulting from pattern application. Pattern relationships are illustrated in Figure 9-5, and discussed in more detail in the “relates to” attribute of each pattern.

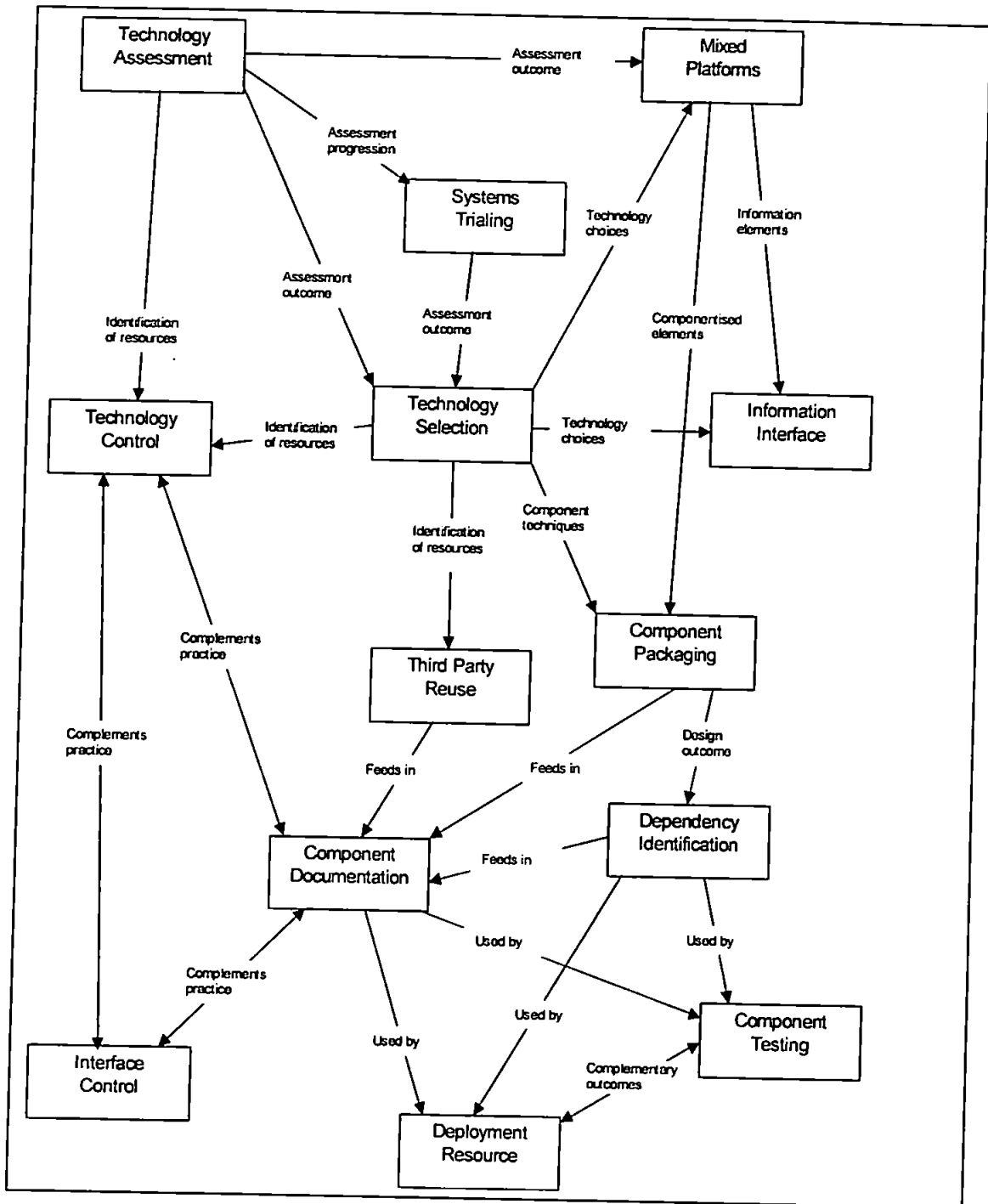


Figure 9-5 – Pattern relationships

9.3.3 Patterns for the adoption and use of component technologies

9.3.3.1 Technology Assessment

Applies to

Manager - In order to ensure correctly functionality at the start of the project

Developer - To carry out trials prior to project execution

Abstract

Do not assume functionality – ensure technologies can perform what is required through trialing prior to project execution.

Problem

Component technologies are still very new compared to other development technologies. Frequent releases and bug fixes can adversely affect a component-oriented project (see Technology Control pattern). In addition, there exists the possibility of expected or assumed functionality not being realised by an implementation. There may also be compatibility issues in the use of different technologies. For example, the difference between CORBA standards and implementations raises a problem. Implementations differ and provide enhancements to the expected standard implementation, leading to incompatibilities

An assumption that can be made by managers and developers is that if component technologies follow the same standard, compatibility will be guaranteed. Experience has shown that this is not necessarily the case.

Example/experience

Case study A provided a number of examples related to assumed functionality and compatibility with CORBA systems. Problems occurred in two different ways:

- **Interfacing objects developed with different CORBA systems:** In the case of project A, two CORBA implementations were chosen, each suitable to the respective target hardware operating system. As both implementations claimed to be IIOP compatible, it was assumed that the IIOP functionality could be used to interface the different objects. When trialing was attempted with the two different implementations, it was discovered that the objects were not compatible. This problem, compounded with the problem on communications overheads (see *Systems Trialing* pattern), resulted in the development of a bespoke inter-ORB protocol.
- **Interfacing objects developed using different language mappings within the same CORBA implementation:** Greater concern arose from the discovery of incompatibility between objects developed using different language mappings (in this case, Java and C++). This caused a greater problem because, in contrast to the interoperability issues between different CORBA implementations, there was no trialing of technologies to test this issue prior to implementation. It was assumed that the objects would interact (as was claimed by the vendor) and therefore no trialing was necessary. Consequently, discovery of this issue was not made until integration testing, which had an extremely detrimental effect upon development schedules.

While case study A provides us with a lot of evidence in a single instance of the need for technology assessment, this is not an isolated case. Survey responses have further highlighted

problems with technologies, with respondents commenting on the lack of cohesion between expected functionality and implementation and also problem reconciling concepts with technologies.

Solution

The trialing of technologies against project requirements prior to commencing the project, or as a parallel activity during design activities, should ensure that the technologies perform in the correct fashion or, if not, enable the identification of problems without impacting upon project schedules. Another benefit from getting development personnel to carry out technology trials prior to implementation is that it enables familiarisation with technologies before they are used within the confines of a project schedule.

Related to

This pattern focuses upon ensuring the correct functioning of technologies and effective development within a project. It complements the *Systems Trialing* pattern which is concerned with ensuring the chosen hardware and software systems can cope with the introduction of component technologies, and the *Technology Control* pattern, concerned with the efficient use of technologies throughout the project. As a result of the practices that form the technology assessment pattern, there could also be influence upon both *Technology Selection* and *Mixed Platforms*.

9.3.3.2 Mixed Platforms

Applies to

Manager – in the selection of technologies in a project

Architect – in determining the choice of technologies for different aspects of the system

Developer – implementing the designed functionality in the chosen technology

Abstract

Use component technologies to their strengths. Do not assume they will solve all problems.

Problem

Most industrial literature related to the adoption of component technologies (for example, [33],[28]) insists that the successful use of component technologies can only come from an organisational embracing of the technologies and a total commitment to their use. Component technologies should replace existing development techniques, organisations should develop component repositories, and developers who produce components will be distinct from those who assemble applications from the components.

A problem facing managers with this approach is, firstly, the level of risk involved. With no proven record in the use of component technologies, can they justify replacing other proven approaches with the new techniques? Additionally, they must face the task of re-skilling – ensuring architects and developers have the requisite skills to be able to carry out a component-oriented project successfully – while still having to commit resources to existing development projects.

Thus they might prefer to phase introduction of component techniques into the development platform, but by doing so they go against current thinking in the area, risking wasted effort and a failed adoption.

Example/experience

Our own experience includes a wholehearted component-oriented project and one that used component-based techniques in tandem with object-orientation and Internet technologies. It indicated that a complete embracing of component technologies does not necessarily lead to a successful project. In Case Study A, the approach was for a 100% component-based solution, making component technologies the underpinning technology for all software development. The hope was to ensure effective interoperation of system elements and provide network transparency. While transparency was achieved, and a level of interoperability was achieved, the pure component approach also introduced over complexity in some areas. For example, core communication functionality was implemented in component form, introducing unnecessary dependencies between clients. Another problem came from a component-based approach to application interfaces that meant GUI development had to interweave two event loops (one for GUI events, one for CORBA events) into the same application.

In case study B, component-orientation was combined with other techniques such as object frameworks and Internet technologies – the most suitable approach was used for each aspect of system development. For internal system functionality, an object framework encapsulated core business functionality. Component techniques were used to interface third party software, and to encapsulate internal business activities. The communication of information between products was achieved using Internet technologies. The resulting development platform, as discussed in section

6.5 demonstrated that component techniques can be used in tandem with other technologies and, in this case, resulted in a far more flexible, and effective, development process.

Solution

If component-orientation is viewed as a development technique, rather than a change in the philosophy of software development, it can complement other, potentially more stable, development technologies.

A mixed platform can have two advantages – firstly, and most importantly, it enables the use of different software technologies to their strengths. However, it also enables the chance for low risk adoption of new technologies, of which component-based techniques could be part.

Related to

The Mixed Platform is related to *Information Interface*, *Technology Selection* and *Component Packaging*, in that they do not uncritically adopt the belief that component-orientation should be the sole technique for software development. It also complements *Technology Assessment*, by providing a way of acquiring skills to underpin component-orientation.

9.3.3.3 Systems Trialing

Applies to

Manager - Ensuring chosen hardware and operating systems can cope with the load of component technologies before project commencement.

Developer - Carrying out trialing prior to the project commencement.

Abstract

Component technologies place additional load on hardware and operating systems. Ensure that selected platforms can cope with this load prior to project commencement.

Problem

Component technologies typically add extra processing overheads to implemented systems. As well as requiring processes in which to execute component instances, the component standard and additional services are themselves implemented as low-level processes. Therefore, component-based systems will generally require more powerful hardware. Additional issues may result from non-optimal support in current operating systems.

Example/experience

Case study A provides an example of the problems that can be experienced due to the additional requirements of a component-based platform. Development work was generally carried out in isolation with small subsets of the complete system being used to unit test component functionality. Testing on a scale approaching full system size was not attempted until integration activities were carried out. Once a full set of objects was loaded onto mobile terminals, it was discovered that the terminals could not cope with the number of objects required to be resident in

memory at a given time. Excessive processing requirements also affected the performance to the point where terminals would hang. Unexpected issues led to already scheduled resource being expended investigating the problem. It was not at all clear whether system lock ups were a result of an error in a component or the component technologies themselves.

Once component technologies had been identified as the cause, loading onto more powerful terminals cured the problems of system lock ups. However, execution was still slow as a result of the load placed on the system by the component platform. This adversely affected trial results, meaning that some aspects (attempts at real time communication) of trialing were not possible.

Solution

In a similar approach to that suggested in the *Technology Assessment* pattern, it is worthwhile putting resources into trialing component technologies on the proposed target systems at the outset of the project. While this may have a minor impact on start dates or resource allocation at the beginning of the project, it could avert costly setbacks if problems are not identified before integration activities. As with the *Technology Assessment* pattern, resource invested at the start of the project in investigating the use of the component technologies can also reduce implementation time as developers will gain experience and knowledge using the technologies.

Related to

The Systems Trialing pattern complements the *Technology Assessment* in seeking to prove the correct functioning of all technologies prior to the start of a project. As an outcome from the practices defined within the systems trialing pattern, *Technology Selection* could also be considered synergistic.

9.3.3.4 Technology Control

Applies to

Manager – identifying the need for control and scheduling trialing.

Developer – ensuring control is held over technologies used

Abstract

The control of technologies used is essential for a successful component-oriented development project.

Problem

Component technologies are still very new in comparison to other development techniques. As such, even more so than with other technologies, there are frequent new releases with new features, bug fixes and enhancements. A common behaviour among software developers is to get the latest version of any piece of development technology as this will be (it is assumed) the best. The problem the manager faces is being able to determine which of the new releases (if any) of each technology are required by the project and whether the new release should be applied to the project. If it is, there may be problems such as backward compatibility. Clearly, the manager needs to ensure that all developers on the project are using the same version of development tools to ensure compatibility among tools. Integration testing is a complex issue in component based systems, and bug tracing can result in tracking through numerous components. It is very important to avoid further confusion resulting from incompatibilities in, for example, the code generated by different compilers.

Example/experience

The culture of “newest version is best” among developers was demonstrated effectively within case study A. The problem in this project was compounded by the distributed nature of the project. It was difficult for managers to get direct control over developers as they were in different geographical locations. A project specification of compiler and technology versions was issued at the start of the project, but there was little control over tools usage once the project had commenced. The complex nature of the project resulted in the use of advanced feature of CORBA technologies. Unsurprisingly, this type of use exposed flaws in the CORBA implementations that resulted in the need for bug fixes from the relevant CORBA vendors. However, this was not carried out at a project level – individuals who discovered problems tended to contact vendors independently, and received individual bug fixes. This resulted in numerous versions of technologies being used to develop the project components.

Problems identified as a result of differences in technologies used included problems integrating the CORBA platform (which was traced to a discrepancy in the minor version number of compilers used) and inter component communication (traced to different CORBA implementations – a conflict between single and multi threaded versions).

Solution

As with interface control, standard change control and communication processes are effective for controlling component technologies. Any new version or technology introduced into the project should come only with official approval and must be communicated to all project personnel. The essential aspects are to ensure that all development personnel are using the same version of technologies and that they are compatible.

Related to

The pattern relates to the family of patterns related to version and change control, providing discipline over variables within the development project, in particular, *Component Documentation* and *Interface Control*. Additionally, we also have issues related to the technologies used for the component-orientation project and, as such, we can also relate this pattern to *Technology Assessment* and *Technology Selection*.

9.3.3.5 Technology Selection

Applies to

Manager – selecting technologies for project development

Abstract

Technology selection should reflect project and organisational requirements, not necessarily the latest fashion in component technology.

Problem

Component technologies are an evolving field that is affected by constantly developing standards and emerging technologies intended to further improve development productivity, application scalability, application integration, etc. Coupled with this continued evolution are the market forces that drive development of the field – component vendors are looking for market share and dominance in an emerging field. Therefore, when making decisions regarding whether to use component technologies and if so, which technologies to use, managers are faced with a bewildering amount of information and hype. This information overload can reduce objectivity in the selection of technologies, resulting in selections that do not reflect the needs of the organisation and the resources available.

As discussed in the *Mixed Platform* pattern, a common misconception in the field is that component-orientation can only be successful if it is wholeheartedly embraced, replacing all previous development technologies. Experience reflected in the *Information Interface* pattern has shown that whereas component technologies excel in the distribution of functionality, the integration of different functional packages (i.e. components) and the benefits of software reuse

that such integration affords, they do not in fact provide an optimal solution to the transfer of information within networked systems.

Example/experience

The problem defined in this pattern may seem to be somewhat obvious – of course technologies should match requirements and experience within an organisation.

However, evidence from the case studies suggests that while one might hope such things would be apparent to an organisation, in reality, this is not always the case. Case study A followed good practice in carrying out an assessment of their requirements for a distributed software system, but it suffered from being very much literature based. Also, and crucially, it was difficult to assess against their requirements, as these were, at that stage, unclear to the project personnel themselves. The result of this lack of rigour in technology selection impacted upon the entire development process.

Case study B had a more considered approach to choice of technologies. Decisions were informed by discussion with developers, both internal and external to the organisation that had used the different technologies being considered. The selection of any technology where organisation personnel had no experience was subjected to trialing prior to project use. This enabled further evaluation of the technologies, provided the opportunity for personnel to get experience with them, and enabled an assessment against project specific requirements.

Survey responses strengthen the opinion that requirement analysis should consider technology selection. Many respondents had trouble learning about component-orientation, with technologies being the major issue in the learning process. Additionally, respondents showed an average time

to feel comfortable with component technologies of approximately 12 months. Without effective requirement analysis that includes technology trialing, it is likely that developers will be working on live project while still not feeling comfortable with the technologies they are using.

Solution

A solution to this problem lies in effective requirements analysis that considers:

- **Project requirements:** Does the project suit a component-oriented approach and is it applicable to all areas of the project?
- **System platforms:** The underlying organisational platforms should play a large part in the choice of component technologies. A core UNIX platform would be far better suited to a CORBA approach, whereas a Windows platform would be better suited to DCOM.
- **Resource available:** In the event that existing personnel do have experience in certain technologies, it should be exploited if possible.

If a component based approach appears appropriate, and it is felt that selected component technologies might be effective, the *Technology Assessment* pattern should be considered, to further confirm suitability and present the opportunity of personnel training.

Related to

The *Technology Selection* pattern provides a high level view of the reasoning underlying the choice of component technologies. *Technology Assessment* and *Systems Trialing* contribute to the overall approach to assessing organisational and project need. Additionally, patterns related to the exploitation of component technologies to their strengths, such as the *Information Interface*, *Mixed Platform* and *Component Packaging* can all contribute to Technology Selection.

9.3.3.6 Information Interface

Applies to

Architect – in designing the system

Developer – in providing an efficient implementation of a design

Abstract

The information interface handles the distribution of information between elements in a distributed system. Such interfaces do not suit a component-based approach.

Problem

Within a distributed system, there are two primary aspects of distribution – functionality and information. The distribution of functionality relates to the spreading of processing requirements across a network. In order to be able to share functionality between elements, a *functional interface* needs to be defined between them. The distribution of information relates to the communication of structured data among elements within the distributed system. While the passing of information may result in an element carrying out some processing requirement, it is not the communication itself that effects the call. In order to be able to share information between elements, an *information interface* needs to be defined between them.

Component technologies provide the ideal environment for the distribution of functionality. A functional interface can be defined using the interface definition language, and functionality implemented in the component technologies aid location, communication and marshalling. However, the communication of structured information introduces communications capacity penalties that can be increased by the use of component technologies. While an information

interface can be implemented using component technologies, to do so can introduce unwanted complexity and also adversely affect execution speed due to the communications overhead.

Example/experience

Case study A featured a total adoption of component technologies, implementing all elements as CORBA objects and using a CORBA platform for the communication of both functionality and information. This was a very effective strategy for distributed functionality. However, it became apparent that the structures being developed for the communication of information were becoming very complex, due in part to the complications of the language mapping of the component standard. Further problems were discovered when executing the system, whose performance suffered for having to pass large information structures using CORBA.

Case study B adopted a more conservative approach to the use of component technologies, exploiting a component approach when distributing functionality across their network, but using Internet technologies for the communication of information. Information interfaces were defined as XML Document Type Definitions. Information took the form of structured XML based upon this type definition. This provided a simpler and more efficient solution. A perceived problem of mixing component techniques with other development technologies did not materialise.

Solution

Use different technologies to their strengths. Component technologies provide effective mechanisms for the distribution of functionality where passed calls do not contain more than function arguments. However, language mapping and communication overheads can result in overcomplex solutions to the communication of information. Techniques such as XML are specifically developed to structure information and, when used on top of simple Internet

protocols, can provide much simpler solutions. Type definition as specified within the XML standard can then be used to specify information interfaces in the same way that IDL is used to specify functional interfaces.

Related to

The *Information Interface* pattern relates to other patterns addresses the optimal use of component technologies. In particular, it compliments the *Technology Selection* and *Mixed Platforms* patterns.

9.3.3.7 Component Packaging

Applies to

Architect – identify the system elements that should be component based, and determine the granularity of component packaging.

Abstract

Do not assume that everything should be implemented as components. Those elements that are components should be of a size that lends itself to effective implementation.

Problem

It is easy to assume that, if component technologies are to be used, everything should be wrapped into component packages. This approach can lead to unnecessary complexity. As with all software technologies, component techniques should be used only where appropriate – the technologies are tools to aid implementation and reuse, not to drive the system realisation.

Additional issues arise from the packaging of functionality in component form. A common question raised regarding both component- and object-orientation is the granularity of classes that provide functionality. However in component-orientation, the greatest problem lies not in the size of component classes, but the scale of the packages in which they are distributed. In order that a class instance is created, the associated component package has to be loaded into memory. Therefore, system efficiency can be adversely affected by individual packages for each component class.

Effective reuse depends on the identification of dependencies among components (see *Dependency Identification* pattern). The *reuse potential* of a component can be adversely affected

by over-dependency as, obviously, a component cannot be reused without the components upon which it is dependent.

Example/experience

This problem also belongs to the set of issues related to the need for adequate knowledge of component techniques when addressing requirements analysis and system design. Our own experience draw from different approaches to the analysis of requirements, with correspondingly different outcomes.

Case study A focused initial design on the transformation of all requirements into system components. As delegation¹³ was used extensively an extremely dependent architecture resulted. In terms of the packaging of functionality, the consequences of this approach were twofold. Firstly the complexity of implementation was increased, and as a result, the processing overhead on the mobile clients was extremely high. Secondly, the potential for reuse of any Case Study A component was low. It was very difficult to isolate a component that could perform a function without delegating some aspect of functionality to another component.

Case study B's more pragmatic and informed approach to the packaging of functionality is also discussed in complementary patterns. Functionality was not always packaged in component form. Inter-related functionality concerned with the internal functionality of the domain in which the organisation existed was implemented as an object framework. As core functionality was an expectation within any software packages developed by the organisation, this approach was

¹³ Delegation is the mechanism by which an object or component draws on the functionality of a different object or component in order to fulfil their own functional request.

deemed most suitable. Functionality external to the core domain, which would be required to differing degrees by different software packages, was developed in a component form and implemented within a single component package. Third party software, which was also used for functionality outside of the organisation's own domain, was also component based ensuring ease of integration.

In relation to the scale of component packaging, this point was most effectively demonstrated in Case Study A, in which CORBA objects were generally contained in single object processes. Therefore, in most cases, the creation of a new type of object required another process to be loaded into memory. This could result in a large number of processes being resident in memory in order to achieve simple functionality. In some cases this resulted in an overload of the hardware and system crashes on the small client terminals.

Solution

The solution to the problems of component packaging and reuse can only result from careful consideration of system needs and experience in the design and implementation of component based systems.

On the basis of our own experience, we can draw a number of conclusions:

1. Component technologies are particularly effective at enabling parts of the system to be reused in other projects, distributing a system over a network, and exploiting existing in-house and third party software.
2. Aspects, such as system-specific user interfaces and processing peculiar to individual elements of the system, derive no benefit from being componentised.

3. Another area in which component technologies do not necessarily offer optimal solutions is in the distribution of information.
4. Complementary functionality should be encapsulated within a single package, in order to maximise reuse potential (through the reduction of dependent packages) and minimise the number of packages resident in memory in an executing system.. Therefore, it is desirable to have a few packages with a number of component classes, rather than provide each component class in a separate package.

Related to

This pattern complements others that relate to the exploitation of component technologies to their strengths, such as *Technology Selection* and *Information Interface*. *Dependency Identification* can be considered synergistic to this pattern in that it also guides the architect in the identification of components and their scale of packaging. *Third Party Reuse* can also be considered synergistic for the same reason.

9.3.3.8 Third Party Reuse

Applies to

Manager – selecting appropriate technologies to enable reuse.

Architect - identifying of areas in which third-party components can be employed to good effect.

The architect can also identify specific third party instances.

Developer - interfacing third party resources to system functionality.

Abstract

Exploit the interoperability benefits of component technologies to draw from the resources of others and focus your own developer effort.

Problem

A constant problem within the software-engineering domain is the problem of recreating existing functionality within a new setting. Numerous examples of similar functionality can often be found, especially with software from the same domain. Some examples of domain reuse can be seen in the area of user interface development, where common toolkits for core functionality (i.e. dialog boxes, buttons, etc.) are available for other developers to reuse.

However, the traditional problem with third party reuse is the learning curve when familiarising oneself with others' software, as a result of source code reuse and non-standard interfaces. If the effort required to interface third party software to in house software products is too high, reuse is not a viable option.

A purported strength of a component-oriented approach is the level of software reuse it affords. Standard interfaces and interoperability protocols should enable a higher degree of reuse than that afforded by other technologies. This pattern centres on the problems of third party reuse and considers whether component-orientation does in fact enable a more effective reuse strategy.

Example/experience

Case study B involved a lot of third party reuse in the implementation – primarily because, as an SME, they could not afford the developer effort outside of the organisational domain. In their case, such an approach successfully enabled complex functionality to be introduced into their applications with little developer effort. The use of component standards for the integration of the components into the applications aided in reducing the amount of integration effort required. In most cases, standard component packages could be easily integrated into their software with no concern for the specific implementation.

We can also identify the benefits of component-orientation for software reuse from the practitioner survey, where over seventy percent of respondents agreed to some degree the component-orientation made software reuse easy. Further analysis of the responses identified differences in opinion depending upon the selected component technologies – COM reuse seemed to be easier to achieve than CORBA reuse. This also reflects our own findings, Case Study A achieving a low degree of reuse with CORBA technologies, while Case Study B achieved a high level of reuse using COM techniques.

Solution

Third party reuse is a powerful way to exploit another developer's domain knowledge to help meet our own requirements. The use of a component standard makes integration far easier than is

possible with, for example, an object framework, where interfaces are non-standard and objects have to be compiled into the main application. If the third party software is written to the same component standard, component assembly tools can be used to further simplify integration.

Third party components are especially beneficial outside the specific area of the organisation, or in a horizontal domain, such as graphical user interfaces or network communication. In these cases, while it is possible to use in-house developers to achieve the required functionality, buying in can save time and effort. The available resources are probably better utilised implementing domain specific components.

Third Party reuse can only be effectively achieved through the combined effort of project personnel – the choice of platform will affect the degree to which third party reuse is available. The identification of areas for third party reuse is essential to focus developer effort upon novel aspects of the system, and the selected third party products have to be successfully interfaced into the system, making correct use of the component technologies.

Related to

This pattern relates to *Technology Selection*, as it will guide reuse strategy. *Component Documentation* is synergistic to Third Party Reuse as it should reflect the different elements of the system, distinguishing those developed in-house with those from third party sources.

9.3.3.9 Dependency Identification

Applies to

Manager - Relies on the architect to identify dependencies that must be taken into account in development schedules

Architect - Identifies dependencies in the system structure and advises managers accordingly

Abstract

Components which client and other component developers require in order to test their own developments must be scheduled accordingly.

Problem

Within a component based system, there will invariably be components upon which others depend to be able to carry out their function. We are faced with a tension between two philosophies within the field of component-orientation – firstly, we have stated that an interface definition in a contract between component server and component clients. With that definition it is possible for the client developer to work independently of the component developer, knowing that the client and component will integrate as both have worked from the same interface specification. However, we must also acknowledge the need for the client developer to unit test their own work prior to integration. While code walkthroughs and comparisons with design documentation will aid in this testing, effective unit testing can only be carried out with dependent components in place. Therefore, they are dependent on delivery of the implemented component in order to ensure effective testing. Without the dependent components, can the developer state that their client/component has been fully tested prior to integration?

Example/experience

Case study A defined a layered architecture where communication passed through several components, each performing some function before delegating responsibility to the next component layer. This architecture resulted in a large number of inter component dependencies. Development of components was generally undertaken in isolation with reliance placed upon interface definition and other design documentation. While some dependencies had been identified, this was at a high level – it was acknowledged that the CORBA platform needs to be in place for testing, and was scheduled ahead of component development. However, there was no dependency identification within the layered component architecture itself. All component development was scheduled for commencement and delivery at the same time.

A requirement on project managers was that developers guaranteed unit testing prior to delivery for integration testing. As dependent components were not available for unit testing, a lot was carried out with dummy components – simple implementations that would return expected values. Components were delivered for integration as “tested as possible”, but without having been tested against the real dependent components.

The inevitable outcome of this approach was an extremely problematic integration process. Many components did not function as expected, and had to go back for further refinement.

Solution

The solution to this potential problem is straightforward – architects of the system are required to identify dependencies and liaise with project managers scheduling development. Scheduling should ensure that those components upon which others rely are available prior to integration. A staggered implementation plan should account for dependencies and therefore aid testing

activities. An additional benefit of this planning is that component developers will be able to identify problems with interfaces prior to them being used by the client developer. However, this solution assumes a simple hierarchical dependency graph, alternative strategies would have to be used for more complex dependencies.

Related to

Dependency Identification complements both the documentation process and post-implementation activities. As such it can be considered synergistic to the *Component Documentation*, as it will affect the documenting of the design process. *Component Testing* and *Deployment Resources* can be considered to complement the *Dependency Identification* pattern as these will be directly affected by effective dependency identification. Finally, *Component Packaging* complements *Dependency Identification* with relation to the potential for component reuse.

9.3.3.10 Component Documentation

Applies to

Manager – putting in place the facilities for change control and version management

Architect – initial production of documentation and revision of documentation

Developer – feeding back changes in implementation that should be reflected in design documentation

Abstract

Documentation from design activities should reflect the current system state to be of value in implementation, testing and deployment. It must be subject to change control in order to avoid design conflicts.

Problem

The issue of documentation is important in any development project. However, it can be argued that component-orientation brings an even greater requirement for documentation that accurately represents both initial state and current system models. The nature of component-orientation should promote isolated development activities and encourage a greater level of third party reuse, due to the interoperability afforded by the technologies. Effective documentation is therefore essential to keep track of component relationships, interfaces between system elements and dynamic behaviour.

Example/experience

We can draw from all three strands of study to illustrate the issues in component documentation. Firstly, the practitioner survey gave a very positive response to the question “Component

orientation encourages design”, with almost ninety percent of respondents agreeing to some degree. This reflects theory developed from the case studies identifying the importance of design documentation for the development of component-based systems.

Case Study A’s rigid development process specified activities for both the static and dynamic specification of the project architecture, and committed substantial project resources to those design activities. Static modelling used OMG IDL for interface definition and OMT [135] models for internal component representation. Dynamic modelling used SDL for interaction diagrams and generated Message Sequence Charts (MSCs) that proved invaluable for component development. As component developers were carrying out implementation in isolation, in most cases in different geographical locations, good design documentation was essential in identifying external interactions that needed to be implemented. General use of MSCs was invaluable to developers in determining their component’s dynamic behaviours. However, problems arose when developers encountered problems with the design and updated their own version of documentation to reflect changes. In many cases these changes were not communicated to a project wide audience, resulting in different versions of particular aspects of design documentation.

The biggest problem with such unconsolidated changes came during integration and deployment. Integration exercises were the first activities to discover the discrepancies in design documentation. Working from official document versions sometimes resulted in conflicts between intended behaviour and implemented behaviour. This resulted in unscheduled revision and re-implementation activities. The most problematic outcome from these discoveries was a complete revision of documentation, working back from implementation. This meant that the “definitive” set of design documentation was only in place following implementation.

Case Study B also committed significant resources to design activities. Design in Case Study B had the additional complexity of integrating a high degree of third party functionality into the systems, as well as using different development technologies. In this case, design documentation was essential to represent the interfacing of system elements and to keep track of internal components, third party components, object frameworks, information interfaces and platform technologies. A range of design techniques were used – UML for object and component relationships and interactions, either Microsoft IDL or Visual Basic class definitions for interface definitions and XML type definitions for information interfaces. Documentation was more effective for implementation and deployment in this case as iteration in the development process encouraged documentation review that would reflect changes brought about through implementation activities. Therefore, when systems were being deployed, the documentation did reflect the *current* system state.

Solution

Good design documentation is essential in component-based projects to ensure project-wide knowledge regarding all system elements is achieved. Important elements include:

- **Static models:** The obvious part of static models is interface definition, which defines the contract between component and clients. However, equally important is component composition documentation, which should represent the internal construction of the component. This is valuable for integration and deployment activities.
- **Dynamic models:** Defining the interactions between system elements and changes of state resulting from these interactions.

- **Platform model:** Valuable in showing where the different tools used at all levels of the system.

Techniques for the design of object-orientation systems are entirely suitable for component-based development. Some approaches, such as UML now include techniques specific to component based development, such as external interface definition and deployment diagrams that model component packaging. However, what is more important than the choice of technique is that all project personnel understand the documentation methods used.

Finally, but crucially, effective version and change control must be applied to documentation to ensure project wide knowledge of the system state. Changes in documentation are essential if it is to represent current system state, but changes should be carried out through the correct process in order that all project personnel are aware of the changes. As with other elements that require change and version control in a component based project, standard software engineering tools and techniques are suitable.

Related to

With its relationship to version and change control within a component-based project, this pattern is synergistic to both *Interface Control* and *Technology Control*. In its reflection of system state, it also complements both *Component Packaging* and *Dependency Identification*.

9.3.3.11 Interface Control

Applies to

Manager – having to provide the mechanisms for the control

Architect – controlling initial specification

Developer – being responsible for change request and communication when required

Abstract

The interface definition defines the contract between component and client developers. Change of interface definition can have an adverse impact on the project unless rigorously controlled.

Problem

The interface definition (or definitions) of a component provides the means for clients (whether these are simple clients or other components) to access component functionality. It is often stated that the interface defines the contract between component and client developers. A strength of the component approach is that it allows component developers to work in isolation as long as they have the required interface definitions. Using the underlying component technologies, theoretically, all should be able to be integrated as all have used the same interface specifications.

The interface definition is, therefore, the crucial aspect of component development. A change to an interface, being used by client developers, can have disastrous consequence for a project, making integration impossible.

Example/experience

While the pattern may, like some other patterns, seem like an obvious and fundamental concern for the managers of a component based project, evidence from our research would suggest that this is not always the case.

Case study A had an architectural design that led to numerous inter-component dependencies (see *Dependency Identification* pattern). Additionally, a number of standard interface definitions provided type information for information structures passed throughout the architecture. As such, the interface definitions were extremely important.

Unfortunately, in a number of cases, incomplete specification and design lead to changes in functionality. In some cases, there was also an impact upon common type definitions. In some cases, developers felt the need to change interfaces in order to be able to carry out their function effectively within the system. Communication of these changes was generally done on an informal basis, with emails sent out to those developers who were using the interface. Inevitably, as no formal system was in place, developers were sometimes missed off the list to whom the change was communicated, resulting in different developers using different versions of the same interface.

The inflexible nature of the development process used by the project meant that these issues were not discovered until integration testing, and consequently many late changes had to be made. This had an extremely detrimental effect on the schedule.

An assumption that this was an isolated issue case was disproved upon surveying practitioners. A number of respondents also indicated issues related to the version control of interface definition as one of the problems experienced when using component technologies.

Solution

The control of the interface includes two elements of standard software engineering practice:

- **Version control:** Ensure the current versions of the interface are available to all developers, and to ensure that a current version cannot be changed when others are using it.
- **Change control and communication:** Any change should be put through a formal change request and review process. Any changes authorised should be communicated in such a way that all developers are aware of the change and are aware of the new version of the interface.

Standard methods can be applied: component-orientation does not introduce anything that is not addressed by existing version and change control systems.

Related to

The issues of version and change control also affect other elements of the component oriented development project that are addressed in the *Technology Control* and *Component Documentation* patterns.

9.3.3.12 Component Testing

Applies to

Managers – scheduling activities and managing resources

Developers – responsible for carrying out testing strategies

Abstract

Traditional testing activities can be affected by the nature of component-orientation. Technologies aid, but do not replace these activities.

Problem

The conventional split into testing of individual elements and integration testing is equally relevant to component-based systems. However, perhaps even more than with traditional systems development, iteration is desirable. Other patterns within this catalog address the philosophy of working in isolation within a component-based project. The belief is, due to the interoperability afforded by component technologies, as long as design documentation is available and scheduling is able to cope, individual components can be developed in isolation and integrated effectively.

However, it does result in integration becoming perhaps the most problematic activity in the development process – this will usually be the first time that developers can see whether requirements have been successfully transformed into a real system. Component technologies do not of themselves prevent or solve problems with design or implementation, and issues concerned with the interoperability of components, in particular in systems where there is a great deal of inter-component dependency, may not be able to be tested until integration.

Example/experience

In considering the testing approaches used in the case studies, Case Study A suffered from a naivety in their approach to system integration. The assumption that the component technologies would ease implementation was, to a point, justified. However, while the actual inter-component aspects were eased by the technologies, the problems with design, which could not be identified until integration due to a per-component level of implementation, were not identified until the integration activity. This impacted a great deal on the development schedule, as no provision was made for feeding back into the development process.

Case study B adopted a far more cyclical approach to its development process (see Figure 9-4), and did not assume that development could be carried out in isolation, relying on component technologies to ease implementation. With this approach, integration was an ongoing process throughout system construction, and fed problems back into the implementation activity as they arose.

Solution

The overall solution to this problem is awareness and vigilance in the integration process. If there is an over-reliance on component technologies, with developers working in total isolation, there are likely to be problems with integration. Countermeasures should be put in place to deal with the problem. The adoption of an iterative process, such as Boehm's spiral model [19] ensures progressive integration and scheduled risk analysis to identify areas of potential problem prior to full system deployment.

Finally, the impact of integration testing can be reduced through the reduction of inter-component dependencies, which can result in unit testing being a more effective activity within the development process.

Related to

This pattern shares concerns with *Deployment Resources*, regarding the philosophy of the component-based approach. *Dependency Identification* also plays a crucial role in aiding the integration process, as it can greatly reduce the need for inter-component integration.

9.3.3.13 Deployment Resources

Applies to

Managers – for the scheduling of resources.

Architects – to carry out deployment activities.

Developers – to carry out the deployment activities.

Abstract

Adequate resources should be put into deployment activities. Good systems knowledge and good understanding of the technologies involved are needed.

Problem

The deployment of a software system can be problematic as it attempts to integrate system elements into a live environment. Component tools that aid the distribution and registration of components across a distributed system address the technical aspects of the deployment process. Further component services can also aid in system deployment.

However, two problems arise from a component-based approach to deployment. Firstly, in order to use component tools effectively, deployers have to be skilled in the use of the technologies and also the underlying concepts. Additionally, system architects who have an overview of the implementation should be available to advise on construction of the system. Managers should not assume that automation of deployment based on the tools available will render the process trivial.

Example/experience

The issue of deployment was addressed in the survey of component developers, with mixed results. Opinion among developers as a whole was fairly balanced, with a slight bias toward the view that component orientation does make deployment easier. Comparison of experience with different technologies showed no pattern in issues related to specific component implementation. It would seem that many developers have experienced both good and bad outcomes from the deployment of component based systems.

Case study experience was similar. It is certainly true that component technologies do aid the technical process of deployment. However, the smoothing of technical obstacles should not be seen as a reason to under resource the deployment activity.

In Case Study A there was little resource committed to deployment, the assumption was that, as the component technologies would aid in the integration of the system, deployment would be a simple assembly. In reality, it was a problematic area, as it was still identifying problems arising from discrepancies in design documentation. It was also evident that those deploying the system had problems relating to both system components and the technologies in general.

Solution

A solution to the problems of deployment with component based systems lies in the identification of suitable personnel and realistic scheduling for the deployment activity.

A conclusion that can be reached from the problems experienced in this case study is that people with both systems knowledge (understanding how system elements should be constructed and

how they inter-relate) and also developers skilled in the use of component technologies should be on hand to ensure effective deployment.

Scheduling for deployment should also consider the potential for problems, as it is evident that deployment may uncover errors that remained hidden in the system implementation. While integration testing should address the issues concerned with component interfacing and technology issues, deployment may identify behavioural problems if testing has not been rigorous.

Related to

Deployment Resources aims to raise awareness of issue in the deployment of component-based systems. It has relationships with *Component Documentation* and *Dependency Identification* in that such issues will impact upon the relative success of system deployment.

9.4 Summary

In the previous chapter, a pattern-based approach was identified as the most suitable for the development of results from the research programme into a usable tool for organisations wishing to develop knowledge in the adoption and use of component technologies. A pattern language developed from the findings of the research programme forms a body of knowledge that can be used to promote understanding about component technologies. By placing the patterns in a context describing the nature in which they were developed, the learning process is further aided by providing the learner the opportunity to be able to relate their own needs to the origins of the language.

10. Conclusion

The research presented in this thesis aimed to assess the impact that component technologies make upon software development. It also aimed to develop assessment findings into a form that would be usable by practitioners who wished to adopt and use component technologies themselves. In order to develop an effective strategy for these aims, three bodies of knowledge were reviewed:

1. In order to determine an effective strategy for assessment, a review of previous work in assessing software technologies was carried out.
2. In order to effectively understand how the results could be developed, a review of previous work in the adoption of technology was carried out
3. In order to establish a baseline for consideration of research findings, a review of current opinion regarding component technologies was carried out. In contrast to other reviews carried out, this focused upon industrial literature. This was due to the lack of academic work in the area [98] and the industrial drivers in the development of component-orientation.

Drawing from the review of assessment approaches, two techniques were used for the collection and analysis of information related to assessing the effect of component-orientation on software development. Case study research enabled an in depth practitioner-focused investigation of the issues in the adoption and use of component systems within two large-scale projects. These case studies led to the development of theories related to the effect of component-orientation within each case. However, as is common with case study techniques, it was difficult to generalise findings. Therefore, another research approach was used to investigate commonality of experience. Practitioners with leading edge component oriented development experience were surveyed in order to identify common issues and phenomena from the case studies. The

surveying approach guided the identification of core problems in the adoption and use of component technologies and focused the development of results.

The development of results drew from the review of adoption techniques, and additional work considering previous approaches to the packaging of information related to software technologies. A pattern approach was considered the most suitable based upon this review and also the nature of results – theories related to the use of component technologies and a body of experience in their use. However, the pattern approach was augmented by surrounding them in a context that enabled users to be able understand their origin and be able to see what degree of relation there was between the context and their own needs. A part of the context drew from the development of a reference model for component platforms – another tool to aid in the education of component-orientation. The reference model is influenced by the aims of other similar models in that it provides an implementation independent view of a software domain. It also enables a comparison of case studies against defined criteria. Initial industrial feedback regarding the package was positive, highlighting the effective structuring of “best practice” without prescription.

10.1 Research Achievements

In the thesis introduction, we discussed findings by the SWEBOK project (see section 1.2) concerning the problem of ascertaining the effect of component orientation upon the development process. More importantly, the project also stated that it was difficult to understand how to present knowledge regarding component integration, and how that knowledge related to other information within the software-engineering field. The findings from this research programme challenge current thinking regarding the adoption and use of component technologies. While such thinking places great importance on distinct technologies and a total embracing of a component approach, the need for knowledge and understanding of concepts has been highlighted, and also demonstrated the applicability of component technologies in tandem with other development

technologies. Therefore the achievements of the programme can be viewed in terms of progressing understanding related to the adoption and use of component technologies. Thus, the research can be seen to provide some response to the problems the SWEBOK project has identified.

Additionally, we define three distinct outcomes from the research:

1. An assessment of component technologies based upon practitioner focused research. The assessment outcomes provide a detailed analysis of the ways in which component technologies can affect a software development project. A significant aspect of the results is that they contradict a lot of current belief regarding the use of component technologies.
2. A reference model for component platforms that has value in two ways. Firstly, it can support learning by isolating the concepts of component-orientation from implementation specifics. It is also a valuable tool for comparative analysis of approaches to the use of component technologies.
3. A contextualised pattern language intended for use as an organisational learning tool for companies wishing to adopt and use component technologies. The language allows adopters to learn from the experience of practitioners without having to follow a prescriptive route. The language is placed in a context that relates the nature of the research surrounding their development. The context also defines a number of points of reference (for example, development process and component platform) to further aid in the ability of users to relate the language to their own tacit knowledge.

10.2 Research Limitations

In considering the limitations of the research presented in this thesis, we examine the research method and also the research findings.

The research method has enabled an in depth analysis of the affect of component technologies upon two software development projects. The value of a case study approach is that it enables research to be carried out within a practitioner context without interfering with the environment. However, the obvious problem with such a method is the level of generalisability that can be drawn from the findings. It is difficult to determine, based solely on case study outcomes, what findings represent component problems and what represent phenomena. While there is opinion that there is value in learning from phenomena (for example, rare event learning [5]) it is often argued that theories developed from a case study should be tested using a different research technique. It would certainly be difficult to develop the results into a learning tool based solely upon the case study findings, as it would be difficult to focus upon common problems. However, the practitioner survey does address this issue to a degree and has enabled a focus of the development of results.

In considering the research findings, there are potential issues with both the theories developed from the assessment and also the pattern language. The issues with the theories are related to the research method, and discussed above. With respect to the pattern language, it could be argued that some of the solutions discussed within the pattern language draw from classic software engineering issues, such as version control and change management. While this could be considered a problem in the development of knowledge related to a supposed leading edge technique, it should be considered against the aim of the research. This was to assess how component technologies affect software development. A philosophy developing from current

thinking about component-orientation is that it should completely change the way that software is developed by organisations. The research findings, in particular the reference to classic software engineering approaches when addressing some of the potential problems in using component technologies, highlights the fact that while component-orientation undoubtedly does affect the development process, in some cases existing techniques provide suitable solutions.

10.3 Future Work

In further developing the research aims, two aspects of development are proposed:

1. **Further validation of the pattern language:** Initial feedback from industrial use of the package has been very positive. This feedback has been used to refine the package to focus more upon the use of patterns to share experience. Greater use of the package within the organisation should result from an invited presentation to technical management early in 2001. Further refinement should result from this work. However, it would be of value to assess the use of the package within other organisations to examine the transferability of the knowledge presented. The context around the language is intended to allow adopters to determine the degree of relevance of results to their own work. Therefore, future work will disseminate the package further through its use in other organisations and also through the publication of results to date.

2. **Quantitative measurement of component-oriented projects:** The research method chosen for this programme has enabled an in-depth assessment of the impact of component orientation and a richness of evidence drawing from numerous sources. However, the qualitative nature of the evidence does not enable any accurate quantifiable measures related to issues such as development productivity when using component-orientation. Such measurement was not possible in the case studies – to try and introduce measurement programmes to the studies would contradict the independence of practice and assessment.

However, now that an unquestionable effect on software development has been established, there would be value in developing a Quality Improvement Paradigm [11] approach, identifying aspects for measurement and applying the GQM technique to assessment.

10.4 Technology Review

Finally, in concluding this research project, we return to the initial aim – to determine how component technologies affect software development. Our research has unquestionably identified a number of issues that result from the adoption and use of component technologies that contradict existing beliefs regarding component-orientation. What should be considered is what component orientation is trying to achieve – is it a whole change in software development, or is it just another contributing technologies. We can view reuse coming from interoperability as the goal of component standards – a component client should be able to interoperate with a component regardless of location and implementation. To a certain extent our research has identified that practitioners have experienced such benefits from component technologies. However, these benefits come at the expenses of complexity, especially if an organisation wishes to become a component producer. It should also be questioned whether component technologies provide the optimal solution to interoperability. Certainly, our second case study investigated and successfully used alternative approaches. The Simple Object Access Protocol (SOAP), being developed by IBM, Microsoft and DevelopMentor also demonstrates a move away from “pure” component approaches such as CORBA or DCOM.

It is unquestionable that component-orientation does play a part in the future of software development – the interoperability benefits for the component consumer are clear to see when one considers the number of components available to any user via a typical Windows installation. Whatever the level of underlying technology, the interoperability afforded by a component based approach affords large improvements in development productivity. However, as identified from

the research programme, the effectiveness of the technology is related to the level of understanding about it. Therefore, we conclude that whatever the effects of using the technology, effective adoption has to come from the development of knowledge about its use.

11. References

1. Abernethy, T. Munday, C. (1995). "Intelligent Networks, Standards and Services", British Telecom Technology Journal 13, 2. April 1995.
2. Alexander (1979). "The Timeless Way of Building". Oxford University Press.
3. Alexander, Ishikawa & Silverstein (1997). "A Pattern Language". Oxford University Press.
4. Arnold, Robertson & Cooper (1992). "Work Psychology: Understanding Human Behaviour in the Workplace". Pitman.
5. Attewell (1996). "Technology Diffusion and Organizational Learning". In Cohen & Sproull (eds). (1996). "Organizational Learning". Sage Publications.
6. Axelrod, R. (Ed.) (1976). "Structure of Decision". Princeton University Press, 1976.
7. Basili & Green (1994). "Software Process Evolution at the SEL". IEEE Software July 1994.
8. Basili & Lanubile (1999). "Building Knowledge through Families of Experiments". IEEE Transactions on Software Engineering July/August 1999.
9. Basili (1996). Editorial. Empirical Software Engineering vol. 1. no. 2.
10. Basili et. al (1995). "SEL's Software Process-Improvement Program". IEEE Software November 1995.
11. Basili, Caldiera, McGarry et. al. (1992). "The Software Engineering Laboratory – An Operational Software Experience Factory". Proceedings of the 14th International Conference of Software Engineering. ISBN 0-81862-6879.
12. Basili, Selby & Hutchens (1986). "Experimentation in Software Engineering". IEEE Transactions on Software Engineering vol.12 no. 7. July 1986.
13. Basili, Zelkowitz, McGarry et. al. (1995). "SEL's Software Process Improvement Program". IEEE Software November 1995.
14. Batistatos, S. (ed.) (1996). "Component Modelling and Design Guidelines". DOLMEN Deliverable MCD2. <http://www.fub.it/dolmen/>

-
15. Bell Labs (1998). "Organizational Patterns". <http://www.bell-labs.com/cgi-user/OrgPatterns/>
 16. Benbasat, Goldstein, Mead (1987). "The Case Research Strategy in Studies of Information Systems". *MIS Quarterly*, September 1987.
 17. Bernstein, P. (1996). "Middleware: A Model for Distributed Systems". *Communications of the ACM* 39, 2.
 18. Bhattacharjee & Gerlach (1998). "Understanding and Managing OOT Adoption". *IEEE Software* May/June 1998.
 19. Boehm (1988). "A Spiral Model of Software Development and Enhancement". *IEEE Computer*, vol. 21, no. 5. May 1998.
 20. Bonofede, V. (ed.) (1998). "Execution of the Final Trial", DOLMEN Deliverable TRD4. <http://www.fub.it/dolmen/>
 21. Booch, G. (1987). "Software Components with Ada", Benjamin/Cummings. ISDB 0-8053-0609-9
 22. Booch, G. (1994). "Object-Oriented Analysis and Design with Applications, 2nd Edition". Benjamin/Cummings Publishing Company. ISBN 0-8053-5340-2
 23. Booch, G. (1997). "Software as a Strategic Weapon". Rational Software, <http://www.rational.com>
 24. Briand, Emam & Melo (1999). "An Inductive Method for Software Process Improvement: Concrete Steps and Guidelines". Appearing in Emam & Madhavji (1999) (eds.). "Elements of Software Process Assessment and Improvement". IEEE CS Press.
 25. Brooks, F (1987). "No Silver Bullets – Essence and Accident in Software Engineering". *Computer* 20,4. April 1987.
 26. Brooks, F. (1995). "No Silver Bullet Refired. In the Mythical Man Month", 2nd Edition. Addison Wesley.

27. Brown & Wallnau (1996). "A Framework for Evaluating Software Technology". IEEE Software September 1996.
28. Brown, J.(1998) "Software Strategies – The Component Decision", The Forrester Report.
<http://www.forrester.com/>
29. Brown, Malveau, McCormick & Mowbray (1998). "Antipatterns: Refactoring Software, Architectures and Projects in Crisis". Wiley Computer Publishing. ISBN 0-471-19713-0
30. Bruno, G. (ed.) (1996). "Evaluation of Service Architecture Frameworks". DOLMEN Deliverable ASD1. <http://www.fub.it/dolmen/>
31. Bruno, G., Lucidi, F. (eds.) (1997). "Service Machine Components (volume 1) – Support of Personal Communications". DOLMEN Deliverable MCD3vol1. <http://www.fub.it/dolmen/>
32. Bruno, G., Lucidi, F. (eds.) (1997). "Service Machine Components (volume 2) - Application Support". DOLMEN Deliverable MCD3vol2. <http://www.fub.it/dolmen/>
33. Butler Group (1998). "Component-based Development. Application Delivery and Integration Using Componentised Software – Strategies and Technologies".
<http://www.butlerforums.com/cbdindex.htm>
34. Butler Group (1998). "Microsoft Make Hard Work of Distributed Objects!!". Butler Group Newsletter December 1998. <http://www.butlerforums.com/cbdindex.htm>
35. Carrington (2000). "Software Engineering Infrastructure". Appearing in SWEBOK (2000). "Guide to the SWEBOK Stoneman version 0.5". <http://www.swebok.org>
36. Chappell, D. (1996). "Understanding ActiveX and OLE". Microsoft Press. ISBN 1-57231-216-5
37. Chappell, D. (1997). "The Next Wave – Component Software Enters the Mainstream". Chappell & Associates. <http://www.chappell.com>
38. Computer Weekly (1998). "Forget Objects, Use Components". Computer Weekly, 19 November 1998.

-
39. Computing (1998). "Merger would be a Beans Feast". Computing Magazine, 26th February 1998.
 40. CORBAWeb (1998). "The CORBAScript Lanaguage". <http://corbaweb.lifl.fr/CorbaScript/>
 41. Cox, B. (1990). "Planning the Software Industrial Revolution". IEEE Software. November 1990.
 42. Cox, B. (1990). "There is A Silver Bullet". Byte October 1990.
 43. Crossan, Lane & White (1999). "An Organizational Learning Framework: From Intuition to Institution". Academy of Management Review vol. 24 no. 3. 522-537.
 44. Cunningham & Kent (1987). "Using Pattern Languages for Object Oriented Programs". Appearing in Meyrowitz (Ed.) (1987) Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87), Orlando, Florida, Proceedings.
 45. Curtis, Kellner, Over (1992). "Process Modelling". Communications of the ACM, September 1992.
 46. Denning, A. (1996). "OLE Controls Inside Out". Microsoft Press. ISBN 1-55615-824-6
 47. Drouin (1999). "The SPICE Project". Appearing in Emam & Madhavji (1999) (eds.). "Elements of Software Process Assessment and Improvement". IEEE CS Press.
 48. Eisenhardt (1989). "Building Theories from Case Study Research". Academy of Management Review vol. 14 pg. 532-550. 1989.
 49. Emam (2000). "Description of the SWEBOK Knowledge Area Software Engineering Process (version 0.5)". <http://www.swebok.org>.
 50. Eveland & Tornatzky (1990). "The Deployment of Technology". In Tornatzky & Fleischer (eds.) (1990). "The Processes of Technological Innovation". Lexington Books.
 51. Fenton, Pfleeger & Glass (1994). "Science and Substance: A Challenge to Software Engineers". IEEE Software July 1994.

52. Fichman & Kemerer (1993). "Adoption of Software Engineering Process Innovations. The Case of Object-orientation". Sloan Management Review. Winter 1993.
53. Fichman & Kemerer (1997). "Object Technology and Reuse, Lessons from Early Adopters." Computer vol. 30 no. 10 October 1997.
54. Fichman & Kemerer (1997). "The Assimilation of Software Process Innovations: An Organizational Learning Perspective". Management Science October 1997.
55. Forrester Corporation (1996). "The Forrester Report – Internet Computing Voyage", <http://www.forrester.com>
56. Fowler & Patrick (1997). "Experience Using a Benchmarking Workshop to Explore One Route to Practical Technology Introduction". From McMaster et. al (eds.) "Facilitating Technology Transfer through Partnership – Learning from Practice and Research". Proceedings of IFIP TC8 WG8.6 International Working Conference on Diffusion, Adoption and Implementation of Information Technology. Chapman & Hall.
57. Fowler & Patrick (1998). "Transition Packages for Expediting Technology Adoption: The Prototype Requirements Management Transition Package". The Software Engineering Institute Technical Report CMU/SEI-98-TR-004.
58. Fowler & Scott (1999). "UML Distilled". Addison Wesley. ISBN 0-20165-783X
59. Furnell, S. (ed.) (1997). "Definition of Resource Adapters". DOLMEN Deliverable MND3. <http://www.fub.it/dolmen/>
60. Gamma, Helm, Johnson & Vlissides (1994). "Design Patterns". Addison-Wesley.
61. Gergen, K.J. (1995) "Social Construction and the Educational Process". In L.P. Steffe & J.Gale (Eds) "Constructivism in Education". Hillsdale, New Jersey: Lawrence Erlbaum
62. Gibbs W. (1994). "Software's Chronic Crisis". Scientific American September 1994.
63. Glass (1994). "The Software Research Crisis". IEEE Software November 1994.
64. Grimes, R. (1997). "DCOM Programming". Wrox Press. ISBN 1-861000-60-X

65. Haines, Canrey, Foreman (2000). "Component Based Software Development / COTS Integration". Software Technology Review. The Software Engineering Institute.
http://www.sei.cmu.edu/str/descriptions/cots_body.html
66. Herbert, A. (1994). "An ANSA Overview". IEEE Network, January/February 1994.
67. Herbsleb & Grinter (1999). "Architectures, Co-ordination and Distance: Conway's Law and Beyond". IEEE Software September/October 1999.
68. Hoffmann (2000). "Software Component Reuse by Adaptation." Ph.D. Theseis. Cork Institute of Technology, Ireland. February 2000.
69. Hope, S., Hammac, J. (eds.) (1996). "Design Requirements for the Final Trial". DOLMEN Deliverable TRD1. <http://www.fub.it/dolmen/>
70. Hope, S., Reynolds, P., Rhodes, S. (eds.) (1997). "Detailed Design of Final Trial". DOLMEN Deliverable TRD2. <http://www.fub.it/dolmen/>
71. Houdek & Kempter (1997). "Quality Patterns – an Approach to Packaging Software Engineering Experience". ACM Software Engineering Notes, 22, 81-88.
72. Houdek (1997). "Software Quality Improvement by using an Experience Factory". Appearing in Leher, Dumke & Abran (eds.) (1997). "Software Metrics – Research and Practice in Software Measurement". Deutscher Universitätsverlag.
73. Humphrey (1989). "Managing The Software Process". Addison-Wesley.
74. Iona Technologies (1999). "CORBA 3. – The Way Forward for Middleware".
<http://www.iona.com/>
75. Iona Technologies (1999). "Orbix Product Guide". <http://www.iona.com/orbix/index.html>
76. ISO (1984). "Basic Reference Model for Open Systems Interconnection". ISO 7498, 1984.
77. ISO (1991). "Quality Management and Quality Assurance Standards – Guidelines for Selection and Use". ISO 9000-3, 1991.

-
78. ISO (1993). "Reference Model for Open Distributed Processing". Document number ISO/IEC JTC1/SC 21/WG7. International Standards Organisation.
 79. ISO(1997). "Introduction to ISO." <http://www.iso.org/infoe/intro.html>
 80. ITU-T (1993). "Message Sequence Charts (MSC)". ITU Recommendation Z. 120.
 81. ITU-T (1994). "CCITT Specification and Description Language (SDL)". ITU Recommendation Z.100.
 82. Jacobsen, I., Griss, M., Jonsson, P. (1997) "Software Reuse - Architecture, Process and Organization for Business Success". ACM Press. ISBN 0-201-92476-5
 83. Jeff Prosise (1999). "Programming Windows With MFC". Microsoft Press. ISBN 1-57231-6950
 84. Jeffrey & Votta (1999). Guest Editors Special Section Introduction. IEEE Transactions on Software Engineering July/August 1999.
 85. Jennings, R. (1998). "Microsoft Transaction Server 2.0", Sams Publishing. ISBN 0-672-31130-5
 86. Katz & Sapiro (1986). "Technology Adoption in the Presence of Network Externalities". Journal of Political Economy, 1986, vol. 94, no. 4.
 87. Kiely, D. (1998). "The Component Edge – An Industrywide Move to Component-Based Development Holds the Promise of Massive Productivity Gains". Information Week April 13, 1998.
 88. Kirtland, M. (1998). "The COM+ Programming Model Makes It Easy to Write Components in Any Language". <http://www.microsoft.com/com/wpaper/>
 89. Kitchenham (1996-1997). "Evaluating Software Engineering Methods and Tools, parts 1-5". ACM SIGSOFT Software Engineering Notes vol. 21 no.1 – vol.22 no. 1.
 90. Koch, G. (1993). "Process Assessment: The BOOTSTRAP Approach". Information and Software Technology vol. 35, no. 6/7. 1993.
-

91. Kruglinski (1994). "Inside Visual C++ Version 1.5". Microsoft Press. ISBN 1-55615-661-8
92. Kunda & Brooks (2000). "Assessing Obstacles to Component Based Development: A Case Study Approach". *Information and Software Technology* vol. 42. June 2000.
93. Landes, Schneider & Houdek (1999). "Organisational Learning and Experience Documentation in Industrial Software Projects". *International Journal of Human Computer Studies*. 1999. No. 51.
94. Levin, R. (1998). "Java Technology Enters the Next Dimension". Sun Microsystems.
<http://java.sun.com/features/1998/11/jdk.html>
95. Liebowitz & Margolis (1998). "Network Externalities (Effects)". In *The New Palgraves Dictionary of Economics*, MacMillan.
96. Liebowitz & Margolis (1999). "Winners, Losers & Microsoft: Competition and Antitrust in High Technology". Independent Institute. ISBN 09455999801.
97. Linton, Vlissides & Calder (1989). "Composing User Interfaces with InterViews". *Computer*, 22(2). February 1989.
98. Maurer (2000). "Components: What If They Gave a Revolution and Nobody Came?". *IEEE Computer* June 2000.
99. McFeeley (1996). "IDEAL: A Users Guide for Software Process Improvement". The Software Engineering Institute Handbook CMU/SEI-96-HB-001.
100. McIlroy, D. (1969). "Mass Produced Software Components". Appearing in [108]
101. McInnis (2000). *Component Based Development: Concepts, Technology and Methodology*. Castek Software Factory Inc. <http://www.cbd-hq.com/>
102. Microsoft (1997). "Windows DNA – Windows Distributed interNet Applications Architecture". <http://www.microsoft.com/dna/>
103. Microsoft Corporation (1997). "Vertical Industry Specifications Supported in Windows DNA". <http://www.microsoft.com/industry/>

-
104. Microsoft Developer Network (1998). "On Complexity, the Web and the Future of Software Development". <http://msdn.microsoft.com/>
 105. Mowbray & Malveau (1997). "CORBA Design Patterns". Wiley Computer Publishing. ISBN 0-471-15882-8
 106. MSDN (1999). "Lessons from the Component Wars: An XML Manifesto". <http://msdn.microsoft.com/workshop/xml/articles/xmlmanifesto.asp>
 107. Murphy, Walker, Baniassad (1999). "Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect Oriented Programming". IEEE Transactions on Software Engineering vol. 25, no. 4. July/August 1999.
 108. Naur, P., Randell, B., Buxton J. (eds.) (1976). "Software Engineering Concepts and Techniques – Proceedings of the NATO Conferences". Petrocelli / Charter, New York.
 109. OMG (1995). "CORBA: Architecture and Specification. Version 2". OMG 1995.
 110. OMG (1998). "CORBA Service Specification". document number 98-12-09. <http://www.omg.org/>
 111. OMG (1998). "Object Management Group Outlines CORBA 3.0 Features". OMG Press Release. http://www.omg.org/news/pr98/9_9.html
 112. Open Software Foundation (1992). "Introduction to OSF DCE". Prentice Hall, Englewood Cliffs, New Jersey.
 113. Orfali, R., Harkey, D., Edwards, J.(1996). "The Essential Distributed Objects Survival Guide". John Wiley & Sons. ISDN 0-471-12993-3
 114. Pallazo, S. (ed.) (1996). "Mobility Function as and Service Architecture Requirements". DOLMEN Deliverable APD1. <http://www.fub.it/dolmen/>
 115. Papert, S. (1980). "Mindstorms, Children, Computers and Powerful Ideas". Scranton, PA. The Harvester Press.
 116. Paulk et. al. (1993). "Capability Maturity Model, Version 1.1". IEEE Software July 1993.
-

-
117. Paulk, Weber, Curtis & Chrissis (1995). "The Capability Maturity Model: Guidelines for Improving the Software Process". Addison Wesley.
 118. PC Week (1997). "Virtual Press Centre – Ovum in the Press, Components", PC Week, March 1997.
 119. Pfleeger & Menezes (2000). "Marketing Technology to Software Practitioners". IEEE Software January/February 2000.
 120. Pfleeger (1999). "Albert Einstein and Empirical Software Engineering". IEEE Computer. October 1999.
 121. Potts (1993). "Software Engineering Research Revisited". IEEE Software September 1993.
 122. Pressman, R. (1992). "Software Engineering – A Practitioner's Approach. European Edition". McGraw-Hill, ISBN 0-07-707936-1
 123. Raatikainen, K. (ed.) (1997). "Evaluation of Current Communication Technologies in Hypermedia Information Browsing". DOLMEN Deliverable TAD3.
<http://www.fub.it/dolmen/>
 124. RACE (1992). "ROSA, 'The ROSA Architecture, Release Two'". RACE Deliverable 93/BTL/DNS/DS/A010/b1.
 125. Raghavan & Chand (1989). "Diffusion Software Engineering Methods". IEEE Software July 1989.
 126. Raman, L. (1999). "Fundamentals of Telecommunications Network Management (IEEE Series of Network Management)". IEEE. ISBN 0-78033-466-3
 127. Redwine & Riddle (1985). "Software Technology Maturation". Proceedings of the 8th International Conference on Software Engineering. IEE Computer Society Press, California.
 128. ReTINA (1999). "ReTINA – An Industrial Quality TINA-Compliant Real-Time DPE".
<http://www.infowin.org/ACTS/RUS/PROJECTS/ac048.htm>
-

129. Reynolds, P., Brangeon, R. (eds.) (1996). "Definition of an Enhanced Distributed Processing Platform for DOLMEN". DOLMEN Deliverable MPD2.
<http://www.fub.it/dolmen/>
130. Reynolds, P., Brangeon, R. (eds.) (1996). "Definition of an Enhanced Distributed Processing Platform for DOLMEN". DOLMEN Deliverable MPD2.
<http://www.fub.it/dolmen/>
131. Rine & Nada (2000). "An Empirical Study of a Software Reuse Reference Model".
Information and Software Technology vol. 42 p. 47-65.
132. Rock-Evans, R. (1997). "Ovum Evaluates: Object Request Brokers – Winners and Losers". Ovum white paper.
http://www.ovum.com/ovum/white_papers/object_request_brokers.htm
133. Rogers (1995). "Diffusion of Innovations (Fourth Edition)". Free Press. New York. ISBN
134. Rogerson, D. (1996). "Inside COM". Microsoft Press. ISBN 1-572-31349-8.
135. Rumbaugh, J., Blaha, M., Premerlani, V., Eddy, F., Lorenson, W. (1991) "Object-Oriented Modelling and Design". Prentice-Hall International, ISBN 0-13-630054-5
136. Schmidt, Johnson & Fayed (1996). "Software Patterns". Guest Editorial for Special Issue on Patterns and Pattern Languages. Communications of the ACM vol. 39 No. 10. October 1996.
137. Schneider, B. (1998). "Comparing Microsoft Transaction Server to Enterprise JavaBeans – White Paper". <http://www.microsoft.com/com/mts>
138. Schuman & Presser (1996). "Questions and Answers in Attitude Surveys". Sage Publications. ISBN 0-7619-0359-3.
139. Schwandt & Marquardt (1999). "Organizational Learning: From World Class Theories to Global Best Practices". CRC Press LRC.
140. SEI (2000). "Technology Adoption". <http://www.sei.cmu.edu/adopting/adopting.html>

- 141. SEI(2000). "The IDEAL Model". <http://www.sei.cmu.edu/ideal/ideal.html>
- 142. Shaw (1995). "Software Architecture: Perspectives on an Emerging Discipline". Prentice Hall.
- 143. Software AG (1999). "What is EntireX – Uniting the Universe of Network Computing". <http://www.softwareag.com/corporat/solutions/entirex>
- 144. Sommerville (1996). "Human, Social and Organisational Influences on the Software Process". Appearing in Fuggetta & Wolf (eds) (1996). "Software Process". John Wiley and Sons.
- 145. Sommerville (1996). "Software Engineering 5th Edition". Addison Wesley ISBN 0-201-42765-6.
- 146. Spector, L. (1998). "Operating System Upstarts Challenge Windows". PC World August 1998.
- 147. SPICE (2000). "ISO/IEC Software Process Assessment – Part 7: Guide for use in process improvement". Working draft v.1.0. SPICE website <http://www.sqi.gu.edu.au/spice/>
- 148. SPICE (2000). "What Is SPICE?". SPICE website <http://www.sqi.gu.edu.au/spice/whatis.html>.
- 149. Stallings, W. (1999). "Snmp, Snmpv2, Snmpv3, and Rmon 1 and 2, 3rd Edition". Addison Wesley Publishing. ISBN 0-20148-534-6
- 150. Steinen (1999). "Software Process Assessment and Improvement: Five Years Experience with BOOTSTRAP". Appearing in Emam & Madhavji (1999) (eds.). "Elements of Software Process Assessment and Improvement". IEEE CS Press.
- 151. Sun Microsystems (2000). "The Java 2 Platform Enterprise Edition: Overview". <http://java.sun.com/j2ee/overview.html>
- 152. SWEBOK (2000). "Guide to the SWEBOK. Stoneman version 0.7". <http://www.swebok.org>.

-
153. Szyperski, C. (1998). "Component Software – Beyond Object-oriented Software". Addison-Wesley. ISBN 0-201-17888-5
 154. Thomas, A. (1997). "Enterprise JavaBeans – Server Component Model for Java". Sun Microsystems White Paper. http://java.sun.com/products/ejb/white_paper.html
 155. TINA Consortium (1994). "Engineering Modelling Concepts (DPE Architecture), Version 2.0". Document no. TB_NS.005_2.0_94. <http://www.tina-c.org/>
 156. TINA Consortium. "Overall Concepts and Principles of TINA, Version 1.0". Document no. TB_MDC.018_1.0_94. <http://www.tina-c.org/>
 157. Trigila, S. (1998). "White Paper on the Status of Progress of the DOLMEN Audit Demonstration". DOLMEN document TR-FUB50.
 158. Trigila, S., Bonafede, V. (eds.) (1998). "ACTS 036 DOLMEN: Final Report". DOLMEN Deliverable ZMF1. <http://www.fub.it/dolmen/>
 159. Tripp & Magee (1997). "A Guide to Software Engineering Standards". Artech House.
 160. Tzifa (ed.) (1997). "Open Service Architecture for Mobile and Fixed Network Environment – Initial Release (volume 1) – Modelling Concepts". DOLMEN Deliverable ASD2vol1. <http://www.fub.it/dolmen/>
 161. Tzifa, B. (ed.) (1997). "Open Service Architecture for Mobile and Fixed Network Environment – Initial Release (volume 2) – Modelling Concepts". DOLMEN Deliverable ASD2vol2. <http://www.fub.it/dolmen/>
 162. VITAL (1999). "VITAL – Validation of Integrated Telecommunications Architectures for the Long Term". <http://www.infowin.org/ACTS/RUS/PROJECTS/ac003.htm>
 163. Wired (1998). "83 Reasons Why Bill Gate's Reign is Over". Wired 6.12 – December 1998.
 164. WWW Consortium (2000). "Extensible Markup Language (XML) 1.0 (Second Edition) - W3C Recommendation". <http://www.w3.org/TR/2000/REC-xml-20001006>
-

- 165. Yin, R. (1993). "Applications of Case Study Research". Sage Publications. ISBN 0-8039-5118-3.
- 166. Yin, R. (1994). "Case Study Research: Design and Methods. Second Edition". Sage Publications. ISBN 0-8039-5663-0.
- 167. Zelkowitz & Wallace (1998). "Experimental Models for Validating Technology". IEEE Computer May 1998.
- 168. Zelkowitz (1996). "Software Engineering Technology Infusion Within NASA". IEEE Transactions on Engineering Management vol. 43 no. 3 August 1996.
- 169. Zelkowitz, Wallace & Binkley (1998). "Understanding the Culture Clash in Software Technology Transfer". Technical report, University of Maryland, 1998.

A. DOLMEN Brochure

B. DOLMEN Evidence Examples

The following provides samples of evidence used in the DOLMEN case study. Footnotes from the evidence discuss its use. Notes tend to relate specific issues to case study propositions. These are reproduced below:

1. Adopting and using component technologies in software development processes will affect process activities
2. An awareness of the issues involved in the adoption and use of component technologies can ease their integration
3. Component technologies ease the development, integration and deployment of distributed systems
4. Uncontrolled adoption and use of component technologies can have a negative affect upon a development project

Excerpt of interview a DOLMEN project manager

Interviewer: How does CORBA help the DOLMEN project?

Manager: In the project we are trying to bring together different network technologies. This is very new work and we are dealing with a number of different operating systems and management interfaces to achieve this. The CORBA platform enables our developers to concentrate the efforts on achieving DOLMEN requirements and not worry about network programming or operating system integration. In interoperability afforded by a CORBA

approach is very important to us – without it we could not achieve our aims within the project schedule.¹⁴

Interviewer: You have a rigid development process compare to current thinking in software engineering. Why is this?

Manager: Telecommunication software is not like other software development. We are primarily concerned with interfacing to hardware, and the software required is very technical. A rigid process like the one we are using reflects telecommunications approaches to hardware development, and is very successful in that context. It enables us to maintain control over the process and ensure that the entire architectural design can be tested prior to implementation.

¹⁵

Interviewer: How does software reuse figure in your strategy for software development

Manager: Software reuse is very important to the project. Being TINA compliant means that we aim to share our software with other TINA compliant projects. At the same time, we hope to be able to reuse components developed by TINA compliant projects to save us time in our own software development. Again, a CORBA approach enables us to address reuse without

¹⁴ Related to propositions 1 & 3, there was an assumption within DOLMEN that CORBA would ease development with little front-loading of effort.

¹⁵ Related to proposition 1, the assumption that telecommunications software was different to “other” software resulted in a development process that was more suited to hardware implementation and one that did not consider the issues that a CORBA approach may introduce.

having to concern ourselves with the technical requirements – these are achieved through the CORBA platform.¹⁶

Sample Email 1

From: XXX
Sent: Monday, October 06, 1997 12:03 PM
To: XXX
Subject: DOLMEN Notice: Implementation of Service Machine Components in C++

Dear all,

As discussed in The Hague, KPN is offering a template which will generate the basic structure of a computational object, based on IDL code for the interfaces of the object. The advantage of this will be that all the final code for the service machine will have the same form, therefore allowing it to be put together easier.

So the question is, are you developing any object in C++ for which this template may be used? Please note that the template cannot be used to generate code for the objects which reside on the terminal, due to the fact that Cool Orb is used there, as opposed to ORBIX, which the template is based on. I already have agreement to have the IA, UAH/UAV, USM, SSM, SF, CSM, CC and LNC developed in this way.¹⁷

If you know of any object not on this list but which is being developed in C++, please tell me as soon as possible, as the final list must be known this week.

Regards,

XXX.

¹⁶ Proposition 3 is addressed with this answer – once again, the assumption that CORBA would ease development and integration is demonstrated.

¹⁷ The issue of a template for component implementation was introduced when it was “discovered” that Orbix did not support multiple interfaces to a single component. Related to proposition 4, this highlights a problem with the lack of knowledge regarding CORBA and its implementations prior to use.

Sample Email 2

From: XXX
Sent: 12 May 1998 18:12
To: XXX
Subject: DOLMEN Notice: Final version of IDLs.

Dear All,

The final version of the IDLs have been put on the FUB/Orange servers at ~wpmc/mc7/FinalIDLs.tar.gz and included here.

The following IDLs have changed :

*common.idl :

1. typedef t_ServiceType t_ServiceTypeName; has been removed
2. exception UserNotResponding{}; has been replaced by
exception UserNotResponding{string name};

*commtypes.idl :

1. Due to probs in CoolOrb enum StatusSB {idle, activestatus, suspendedstatus}; is changed to
enum StatusSB {idlestatus, activestatus, suspendedstatus};

2. The exception NonAssociatedSFEP is added

*pa.idl The version within PA_090598.tar.gz has been used

1. void DisplayInvitations() ; has been replaced by
t_InvitationList DisplayInvitations();

*tcsml.idl and also TCSM_CSM.idl

Activate has been changed to Activate_ and the function QueryNFEP is put in.

*Stream_Interface.idl

AddFlow (inout StreamFlowEndPoint endpoint) This is as agreed in MC-KPN24.

*HIB GSS.idl

A new version now exists which also replaces i_HTTP_AccessRequest.idl and i_ProxyControl.idl. Thus these two interfaces have been removed.

*UAPHIB = i_AgentControl.idl

```
oneway void error(in EncodedString err, in EncodedNVPairList params); is
now
oneway void send_error(in EncodedString err, in EncodedNVPairList params);
```

If any changes have occurred in the CMA's IDLs due to Handover, they are not yet available. Also the idls for the new objects have yet to be released.

For this reason the IDLs included here may be updated once more, but no further update to common.idl will now occur.

If these IDLs could be made available on the FUB/Orange server before the end

of this week (Friday 15th May) it would be very helpful as we could then supply Orange complete set of IDLs on Monday 18th May.¹⁸

Due to the above fact, implementers will now only need to supply their code and not the IDLs to Orange.

I hope this will solve any problems, and can you all please ensure that the correct type definitions are include in your code.

I would like to thank you all for you co-operation

Regards

XXX

Excerpt from DOLMEN Working Paper

Mix Java, WWW and CORBA mechanisms

The scheme of interaction between generic components in WWW and Java based implementation can be illustrated by the figure 1.

In this figure, we identify a Java-enabled web browse, a WWW server, an applet store database, a CGI program and CORBA objects.

A user starts a Java-enabled web browser on his machine, enters an URL address and the browser connects him to a WWW server. The typed address should be a description of an Internet

¹⁸ This sample email highlights two issues discussed within the case study, both related to propositions 2 & 4. The fixing of interface definitions did not occur due to problems with designs. As such, numerous changes were made to the definitions. The communication of changes was carried out in this ad-hoc manner, with emails detailing changes to all concerned parties.

resource. Using the browser, he can download and execute applets from the WWW server. The applet can ask the browser to display messages, or different Web pages located on the server. Applets can also GUI to display information and to read inputs from the user.

For sending data from an applet to CORBA objects, we can use a gateway CGI program. The CGI programs can then use CORBA mechanisms to access standard CORBA objects (we tested "C" CGI programs with Orbix objects).

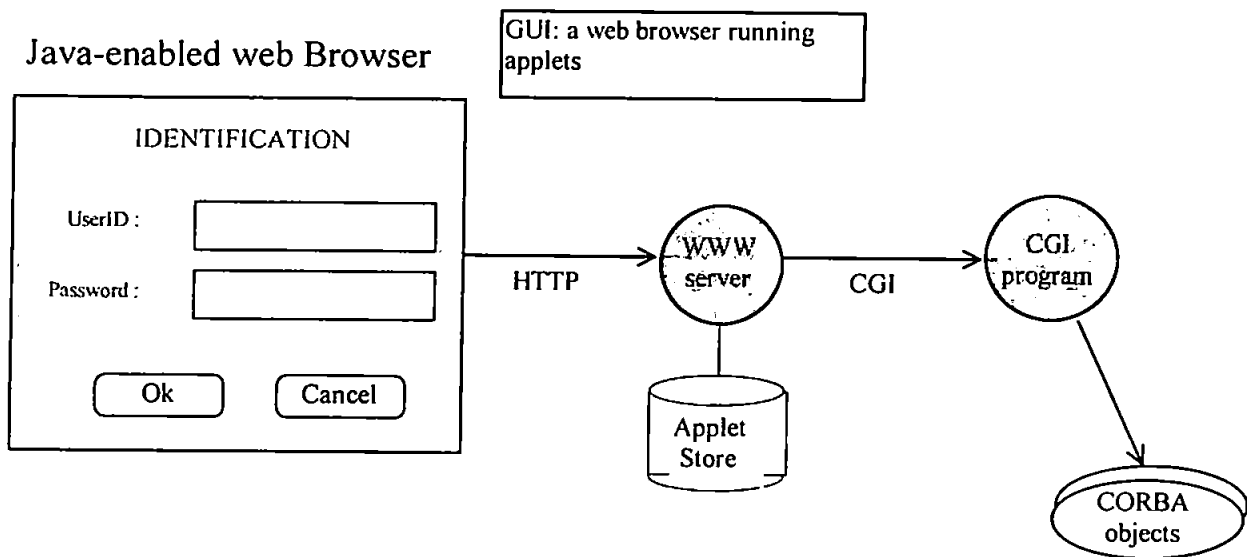


Figure 1-Java to Corba

A direct interaction between the applets and CORBA objects is possible in the specific case of Orbix using OrbixWebIIOP (Internet Inter Orb Protocol). Since Java and C++ languages are close, functionality of an Orbix C++ client can be implemented in a Java program or applet. An OrbixWebIIOP IDL compiler is used to translate the mapping of an Orbix C++ client into Java one. The Java client is then used instead of Orbix C++ client.

Since we are planning to use Orbix in Dolmen, an approach based on the two alternatives to implement the user system can be investigated.¹⁹

¹⁹ Taken from a working paper, this except demonstrates the assumptions related to the use of CORBA implementations within the development of DOLMEN software (and as such, related to propositions 2 & 4). In this case, Java was not considered appropriate for implementation until it was decided to reuse existing WWW software for the browsing application. It was then assumed that the software could interface with the DOLMEN architecture via IIOP, but not testing was done of this assumption prior to integration.

Excerpt from meeting notes (observation)²⁰

Notes from TA Workpackage meeting 12/2/98
University of Helsinki, Finland.

Note - Stream binding functionality should follow working paper MC-TF07.

Note- TCSM is being developed by XXX. Refer to working paper MC-NTU12 for specification.

These design issues relate to working papers, not formal design documents

Application has dependency upon TCSM for stream set up.

Note - User interface should provide options for invites, suspending and closing sessions.

Issue – is there any support for GUI development in CoolORB – C libraries or Tk/Tcl? Need for custom libraries?

Related directly to CORBA implementation, should such problems have been identified during technology selection?

Note - Application startup should determine whether an invite or resumption is being carried out. Passing a session id of the form <retailer>_Audio_<session_no> should signify the resumption of a session. Not session id should mean that the session is new.

Retailer = RetUK or RetFIN

Suspending from the application should carry out the following:

Break stream connection

Suspend session through the session manager reference

Inform suspendee that the session has been suspended via the GUI

Close application

Distinguish between session owner when closing sessions on the GUI. Provide two buttons:

Leave – end call (owner), leave call (other)

End – quit session (owner), leave call (other).

²⁰ Notes from meetings used to document observations regarding the DOLMEN use of component technologies.

Comments related to research aspects were included in *italics*. These meeting notes highlight a number of problems related to design issues and problems with CORBA implementation (propositions 2 & 4 are addressed).

Integration to start 25/5/98

-it will have to be staggered because the software won't be ready in time.

Draws from problems related to the identification of object dependencies and the scheduling of development

Note – a redefinition to the common.idl spec

Enum StatusSB { idle, activestatus, suspendstatus }

Change idle → idlstatus

Changes in specification “ad-hoc” based upon meeting discussion

Note – new TCSM to be distributed to overcome bugs in mobile side stream connection

Issue – what about deleting a stream binding? Should an MSC be generated for this?

Further “ad-hoc” design issues

Note – logging out with sessions running should not be possible as the User Agent should deny a log out request if there are any sessions running.

Issue – what happens if the application crashes? (with relation to session and stream connections)

Issue – What shells should be used for start-up scripts

Outstanding issues in user and service session start-up:

Remove ProfileComp from auduiImpl::Update

Determine QoS rates from session manager

Need to keep track of the user profile in the session manager which can be called after the UpdateProfile in the interface code

When GetSFEP is called by the session manager, this is an indication to set up a stream binding (refer to MSC!).

Add a function in the application to send a TCSM and SFEP ref to the session manager.

Excerpt from Software Development Crisis White Paper²¹

See section 5.5.4

²¹ The excerpt from a white paper regarding the software development crisis from DOLMEN can also be considered an example of evidence from the case. This relates to propositions 2 & 4, in that it contrasts what was expected with what happened with the use of components.

C. Netscient Evidence Examples

The following provides samples of evidence used in the Netscient case study. Footnotes from the evidence discuss its use. Notes tend to relate specific issues to case study propositions. These are reproduced below:

1. Adopting and using component technologies in software development processes will affect process activities
2. An awareness of the issues involved in the adoption and use of component technologies can ease their integration
3. A domain-oriented approach to component development provides a greater degree of reuse than a product oriented view.
4. Similar issues with component-orientation occur when using different technologies from the same field (i.e. Microsoft based, rather than OMG based technologies)
5. Issues in the DOLMEN case study can be avoided through greater knowledge of the technologies involved

Excerpts from an interview with a Netscient director

Interviewer: In your selected approach to developing your software, you don't choose a 100% component approach. Why is this?

Director: Component techniques are still a volatile area. Choosing a technology for our products that uses an approach that could be obsolete in six months is too risky. Do you choose COM or do you choose CORBA, and is CORBA going to survive the MS onslaught? Obviously, we see potential in components, which is why we are looking at their use with in house systems. At the moment it is looking promising. We also see this as a chance to gets some skills in component development without risking our product line.²²

²² This point tie into propositions 2 and 5. There is an illustration of greater caution and less trust of "hard" information that is provided by vendors.

Interviewer: Do you see initial skills as important for a transition to components?

Director: Absolutely. Especially with components – where are the training courses and where are the consultants? It untested waters for ISVs. Do we have any guarantees that an 100% components approach will work, apart from the vendors, and they'll say anything to get a sale. If we can't get a yardstick, we have to do our own assessment before using them. I think we've got some great developers here – Tim really knows his stuff. So, get him to look at components and tell us what he thinks. We can't risk our entire company on the newest hype technology – our investors wouldn't let us!

Interviewer: What strengths do you see from a component approach.

Director: While the encapsulation of internal systems appeals, we see the big benefit coming from third party integration. We are an SME competing with large organisations. Look at TMN – it's a huge undertaking to develop our own suite to interface with TMN systems, but if we could buy in a component set and plug it into our own products – heaven!. On a smaller scale we are making great use of GIS functionality within N-Centre. Having a GeoConcept component has made this very straightforward.

Excerpt from email interview with head developer

Interviewer: How is the GIS functionality embedded into the main application (OLE, ActiveX control, whatever)?

Developer:

Netsigner uses the following external components,

It uses GeoConcept (GIS) via an ActiveX Control

It uses Addflow (Diagramming) via an ActiveX Control

It uses Crystal Reports through OLE Automation

It uses the IE XML COM component (I wish MS would hurry up and release a standalone redistributable!)

It use the ADO COM component (Why don't Oracle release their own OLEDB provider, Why doesn't MS Oracle provider work!)

Anything external to our core functionality really! COM integration is straightforward using VC++ so it seems daft to rewrite existing work.

Interviewer: Why chose XML to distribute information?

Developer:

Have you tried passing big structures in COM! Especially C++! I like coding but not that much. But seriously, the overhead passing information with a component-based approach is too high, its too complex and its inefficient. XML presented an elegant solution – I know its supposed to be web based but it seemed sensible for in house comms too. We're still using TCP/IP, just not

browser technology. Components are still an essential aspect – generation and parsing are both done with them – MS parser and in-house generator.²³

Excerpt from early requirements document

Overview of Netscient Administrative Structure²⁴

Netscient are a software house specialising in the development of simulation and management software for network providers. Their first suite of applications focuses on providing network managers with the means to design and plan networks without having to affect the live network until a new design is acceptable. The planning software must therefore be responsive to organisational limits on networking equipment (for example, being aware of how many customer connections are on a given node). It must also be aware of hardware specific conditions that will affect a network configuration (for example the maximum number of connections to a network card).

While the awareness of company specific parameters can be set into the application by a user, the hardware boundaries can vary greatly:

- between models in the same family;
- between models in different families;
- between different vendors equipment.

Obviously, to accommodate all of these differences in a single application or application suite would result in a huge application which would require constant updating. In order to escape this problem, Netscient came up with the concept of externalizing the behaviour of a specific piece of equipment (termed a *personality* by Netscient), and selling personality sets with the application suite.

Therefore, in terms of the internal development issues, Netscient are faced with a number of problems

How are these personalities described?

How are they stored?

How are new personalities (i.e. new vendors) entered into the personality store?

How are personalities extracted from the store

How can these personalities be managed (i.e. kept to a reasonable level of detail)?

²³ Another illustration of a greater awareness of the strengths of a component approach – a result of trialing work with the technologies. Having a good understanding of the issue involved has enabled a exploitation of components while ensuring that problem areas are avoided.

²⁴ Related to proposition 3, this early document focused upon domain, not product, functionality for the organisation. It was an early requirements spec that influenced a lot of the early system design.

Sample email 1²⁵

From: XXX
Sent: Monday, October 12, 1998 5:21 PM
To: XXX
Subject: Re: TMN Stuff

Hi again,

TMN is a seriously big issue for us. We will have to graft a TMN interface onto N-Center in order to communicate (transfer & receive) information on sales orders, customer provisioning profiles, equipment configurations, billing profiles.

We have joined the Network Management Forum (WWW.NMF.ORG) whose SMART TMN initiative is intended to bring out the process data from an otherwise reservoir of academic standards. This is the route we are likely to take, i.e. applying process-specific data to N-Center.

I think our favoured approach will be to use a TMN converter, probably bought-in. So yes TMN is on the radar screen. But not yet ready to load the missile.

Regards

XXX

Sample email 2 – example of external systems integration

From: XXX
Sent: Thursday, July 22, 1999 5:01 PM
To: XXX
Subject: Cisco integration

XXX

another favour.

The attached message follows a meet and demo to XXX at Cisco. He talked about Directory Enabled Networks, and the 'net' location of Cisco inventory.

²⁵ Both sample emails relate to the issue of third party integration – it illustrates the intention to integrate with outside functionality, and to reuse the software of others. This intention drove a need to exploit component standards for interoperability.

The idea running through my mind was lets get to it and load it as a personality table in N-Center. XXX gave me the following addresses and I've got to the murchiso site and downloaded the attached powerpoint presentation on schema.

Do you think you could give a couple of lines on the relevance of this to our personality tables? Seems neat that cisco do it and keep it up to date and we just download it?

Cheers

XXX

PS file will follow - my PC's run out of memory!

-----Original Message-----

From: XXX

Sent: 20 July 1999 08:20

To: XXX

Cc: XXX

Subject: Re: ** Ping **

It was good to see you and your product, I think we can get some good interest in it and I will start to introduce CAP G and Compaq to the prospect of talking to you.

For DEN schema try

<<http://murchiso.com/den/>><http://murchiso.com/den/>

I found the link via

<http://wwwin.cisco.com/nsmbu/Products/active_directory/ad_main.htm>http://wwwin.cisco.com/nsmbu/Products/active_directory/ad_main.htm

<http://wwwin.cisco.com/nsmbu/Products/active_directory/den.html>http://wwwin.cisco.com/nsmbu/Products/active_directory/den.html

>

>

Regards

XXX

Design documentation sample: Sample information interface definition²⁶

```

<!ELEMENT PROFILE (NODE|SHELF|CARD|CONNECTION|
                  SITE|CURRENCY|STATUSGROUP|ORDER|
                  JOB|PROJECT|TRUNK|CABINET|SVC|VC|
                  LOGICALPORT|PORT|CIRCUIT|CHANNEL|CABLESEGMENT) *>
<!ELEMENT NODE (COMPATIBLESHELF, PARAMETER) *>
<!ELEMENT COMPATIBLESHELF EMPTY>
<!ELEMENT SHELF (COMPATIBLECARD, PARAMETER) *>
<!ELEMENT COMPATIBLECARD EMPTY>
<!ELEMENT CARD (PARAMETER) *>
<!ELEMENT SITE (PARAMETER) *>
<!ELEMENT CURRENCY (PARAMETER) *>
<!ELEMENT STATUSGROUP (MANDATORYPARAMETER, PARAMETER) *>
<!ELEMENT ORDER (MANDATORYPARAMETER, PARAMETER) *>
<!ELEMENT JOB (MANDATORYPARAMETER, PARAMETER) *>
<!ELEMENT PROJECT (MANDATORYPARAMETER, PARAMETER) *>
<!ELEMENT TRUNK (MANDATORYPARAMETER, PARAMETER) *>
<!ELEMENT CABINET (MANDATORYPARAMETER, PARAMETER) *>
<!ELEMENT SVC (MANDATORYPARAMETER, PARAMETER) *>
<!ELEMENT VC (MANDATORYPARAMETER, PARAMETER) *>
<!ELEMENT LOGICALPORT (MANDATORYPARAMETER, PARAMETER) *>
<!ELEMENT PORT (MANDATORYPARAMETER, PARAMETER) *>
<!ELEMENT CIRCUIT (MANDATORYPARAMETER, PARAMETER) *>
<!ELEMENT CHANNEL (MANDATORYPARAMETER, PARAMETER) *>
<!ELEMENT CABLESEGMENT (MANDATORYPARAMETER, PARAMETER) *>
<!ELEMENT PARAMETER (AVAILABLEVALUE) *>
<!ELEMENT MANDATORYPARAMETER (PARAMETER) *>
<!ELEMENT AVAILABLEVALUE EMPTY>

<!ATTLIST PROFILE
    NAME          CDATA          #REQUIRED
    DESCRIPTION    CDATA          #IMPLIED
    LEVEL          CDATA          #FIXED    "2">

<!ATTLIST NODE
    ID             CDATA          #REQUIRED
    VENDOR         CDATA          #REQUIRED
    TYPE           CDATA          #IMPLIED
    SUBTYPE        CDATA          #REQUIRED
    DESCRIPTION    CDATA          #IMPLIED

```

²⁶ An illustration of an interface definition away from component technologies.. This specification was used by both in house and product developers in the development of their systems and proved to be a very successful piece of design documentation.

```

        VERSION          CDATA          #REQUIRED
        NOSHELVES        CDATA          #REQUIRED>

<!-- ATTLIST SHELF
        ID                CDATA          #REQUIRED
        VENDOR            CDATA          #REQUIRED
        TYPE              CDATA          #IMPLIED
        SUBTYPE           CDATA          #REQUIRED
        DESCRIPTION        CDATA          #IMPLIED
        VERSION           CDATA          #REQUIRED
        NOSLOTS           CDATA          #REQUIRED
        AVAILABLESLOTS    CDATA          #REQUIRED
        SLOTTYPE          CDATA          #REQUIRED>

<!-- Mandatory parameters for CARD type
        #SlotNos: A list of slot numbers in which a card can fit
        #SupportedProtocols: A list of protocols (T1, T3, etc.) the
card supports-->

<!-- ATTLIST CARD
        ID                CDATA          #REQUIRED
        VENDOR            CDATA          #REQUIRED
        TYPE              CDATA          #IMPLIED
        SUBTYPE           CDATA          #REQUIRED
        DESCRIPTION        CDATA          #IMPLIED
        VERSION           CDATA          #REQUIRED
        CHANNEL           CDATA          #REQUIRED
        TOTALCAPACITY     CDATA          #REQUIRED
        CAPACITYUNIT      CDATA          #REQUIRED
        CARDTYPE          CDATA          #REQUIRED
        NOPORTS           CDATA          #REQUIRED
        PORTCAPACITY      CDATA          #REQUIRED>

<!-- ATTLIST COMPATIBLESHELF
        ID                CDATA          #REQUIRED
        DESCRIPTION        CDATA          #IMPLIED>

<!-- ATTLIST COMPATIBLECARD
        ID                CDATA          #REQUIRED
        DESCRIPTION        CDATA          #IMPLIED>

<!-- ATTLIST PARAMETER
        NAME              CDATA          #REQUIRED
        DESCRIPTION        CDATA          #IMPLIED
        VALUE              CDATA          #REQUIRED
        MANDATORY          (YES|NO)      #IMPLIED
        FIXED              (YES|NO)      #IMPLIED
        TYPE                (STRING|BOOL|FLOAT|INT|DATETIME|FORMULA)
        #REQUIRED>

<!-- ATTLIST AVAILABLEVALUE

```

VALUE	CDATA	#REQUIRED>
-------	-------	------------

```

<!-- ATTLIST SITE
  ID          CDATA #REQUIRED
  TYPE        CDATA #IMPLIED
  SUBTYPE     CDATA #REQUIRED
  DESCRIPTION CDATA #IMPLIED
  VERSION     CDATA #REQUIRED>

<!-- ATTLIST CURRENCY
  ID          CDATA #REQUIRED
  TYPE        CDATA #IMPLIED
  SUBTYPE     CDATA #REQUIRED
  DESCRIPTION CDATA #IMPLIED
  VERSION     CDATA #REQUIRED>

<!-- ATTLIST ORDER
  ID          CDATA      #REQUIRED
  TYPE        CDATA      #IMPLIED
  SUBTYPE     CDATA #REQUIRED
  DESCRIPTION CDATA      #IMPLIED
  VERSION     CDATA      #REQUIRED>

<!-- ATTLIST JOB
  ID          CDATA      #REQUIRED
  TYPE        CDATA      #IMPLIED
  SUBTYPE     CDATA #REQUIRED
  DESCRIPTION CDATA      #IMPLIED
  VERSION     CDATA      #REQUIRED>

<!-- ATTLIST PROJECT
  ID          CDATA      #REQUIRED
  TYPE        CDATA      #IMPLIED
  SUBTYPE     CDATA #REQUIRED
  DESCRIPTION CDATA      #IMPLIED
  VERSION     CDATA      #REQUIRED>

<!-- ATTLIST STATUSGROUP
  ID          CDATA      #REQUIRED
  TYPE        CDATA      #IMPLIED
  SUBTYPE     CDATA #REQUIRED
  DESCRIPTION CDATA      #IMPLIED
  VERSION     CDATA      #REQUIRED>

<!-- Mandatory parameters for TRUNK type
  #Trunks: List of trunks in the connection
  #VCs: List of VCs in the connection
  #Circuits: List of circuits in the connection-->
<!-- ATTLIST TRUNK
  ID          CDATA      #REQUIRED
  TYPE        CDATA #IMPLIED

```

```

SUBTYPE          CDATA #REQUIRED
DESCRIPTION      CDATA      #IMPLIED
VERSION          CDATA      #REQUIRED
MAXVC            CDATA #REQUIRED
MAXCIRCUIT CDATA #REQUIRED>

```

```
<!-- ATTLIST CABINET
```

```

ID              CDATA      #REQUIRED
TYPE            CDATA #IMPLIED
SUBTYPE         CDATA #REQUIRED
INTERNALDIMENSIONS CDATA #IMPLIED
EXTERNALDIMENSIONS CDATA #IMPLIED
DESCRIPTION      CDATA      #IMPLIED
VERSION          CDATA      #REQUIRED>

```

```
<!-- ATTLIST SVC
```

```

ID              CDATA      #REQUIRED
TYPE            CDATA #IMPLIED
SUBTYPE         CDATA #REQUIRED
DESCRIPTION      CDATA      #IMPLIED
VERSION          CDATA      #REQUIRED>

```

```

<!-- Mandatory parameters for VC type
#hops: List of hops on the VC
#VCs: List of VCs the VC belongs to-->

```

```
<!-- ATTLIST VC
```

```

ID              CDATA      #REQUIRED
TYPE            CDATA #IMPLIED
SUBTYPE         CDATA #REQUIRED
DESCRIPTION      CDATA      #IMPLIED
VERSION          CDATA      #REQUIRED
MAXHOP           CDATA #REQUIRED
MAXVC            CDATA #REQUIRED>

```

```
<!-- ATTLIST LOGICALPORT
```

```

ID              CDATA      #REQUIRED
TYPE            CDATA #IMPLIED
SUBTYPE         CDATA #REQUIRED
DESCRIPTION      CDATA      #IMPLIED
VERSION          CDATA      #REQUIRED>

```

```

<!-- Mandatory parameters for PORT type
#VCs: List of VCs on the port-->

```

```
<!-- ATTLIST PORT
```

```

ID              CDATA      #REQUIRED
TYPE            CDATA #IMPLIED
SUBTYPE         CDATA #REQUIRED
DESCRIPTION      CDATA      #IMPLIED
VERSION          CDATA      #REQUIRED
VCCAPACITY CDATA #REQUIRED
MAXVC            CDATA #REQUIRED>

```

MAXVP CDATA #REQUIRED>

<!-- Mandatory parameters for CIRCUIT type
#trunks: List of trunks on the circuit-->

<!ATTLIST CIRCUIT

 ID CDATA #REQUIRED
 TYPE CDATA #IMPLIED
 SUBTYPE CDATA #REQUIRED
 DESCRIPTION CDATA #IMPLIED
 VERSION CDATA #REQUIRED
 TRUNK CDATA #REQUIRED>

<!ATTLIST CHANNEL

 ID CDATA #REQUIRED
 TYPE CDATA #IMPLIED
 SUBTYPE CDATA #REQUIRED
 DESCRIPTION CDATA #IMPLIED
 VERSION CDATA #REQUIRED>

<!ATTLIST CABLESEGMENT

 ID CDATA #REQUIRED
 TYPE CDATA #IMPLIED
 SUBTYPE CDATA #REQUIRED
 DESCRIPTION CDATA #IMPLIED
 VERSION CDATA #REQUIRED>

D. Component Survey

Component Technologies Survey

This survey is being conducted as parts of a Ph.D. investigation into the impact of component technologies upon software development, in order to develop better practices for their adoption and use.

If you have any other comments regarding component orientation that you feel may help with this investigation, please send them to andy@jack.see.plym.ac.uk

Thank you

Andy Phippen

School of Computing

University of Plymouth

Drake Circus

Plymouth

Devon PL4 8AA

England

About you (optional)

Your name

Your position

Organisation name

Regarding your use of component technologies

1. How long have you been using component oriented software techniques?
2. How long have you been developing software in general?
3. What component standards have you used?
 - ☐ CORBA
 - ☐ COM
 - ☐ DCOM
 - ☐ COM+
 - ☐ EJB
 - ☐ Other (please specify):
4. What component tools and technologies have you used?
5. On how many projects have you used component oriented techniques?
6. On what scale of project have you used component oriented techniques?
 - ☐ Small scale investigation/assessment
 - ☐ Small in house development
 - ☐ Intra organisation development

-
- ☐ Product development
 - ☐ Enterprise development
 - ☐ Pan-enterprise development

7. In what vertical domains did these projects reside?

- ☐ Government
- ☐ Healthcare
- ☐ IT services
- ☐ Manufacturing
- ☐ Retail
- ☐ Telecommunications
- ☐ Banking
- ☐ Construction
- ☐ Education
- ☐ Energy
- ☐ Military
- ☐ Accounting
- ☐ Insurance
- ☐ Legal
- ☐ Media
- ☐ Other (please specify):

Regarding your learning of component technologies

8. How did you learn about component technologies?

- ☐ Industrial course
- ☐ Academic course
- ☐ Reading
- ☐ Practical project
- ☐ Other (please specify):

9. Did you experience problems when learning about component technologies?

- ☐ Yes
- ☐ No

10. If yes, were these problems:

- ☐ Related to concepts
- ☐ Related to technologies
- ☐ Related to differences between concepts and technologies
- ☐ Other (please specify):

If you wish to describe these problems further, please do so below:

11. Did you find the literature about component orientation useful when learning about it?

- ☐ Yes
- ☐ No

If no, why was the literature not useful?

12. Would it have been beneficial to be able to draw form the experiences of others who had used component technologies?

- ☐ Yes
- ☐ No

13. How long did it take before you felt comfortable with component technologies?

Regarding component technologies and the software development process

14. Was integrated component orientation into your development process straightforward?

- ☐ Yes
- ☐ No

If no, what problems did you encounter?

15. Do you believe that component orientation makes software development:

- ☐ Easier
- ☐ Harder
- ☐ Neither easier or harder
- ☐ Other (please specify):

16. If you believe that component orientation makes software development harder, why is this?

If you believe that component orientation makes software development easier, why is this?

17. Given the choice, would you use component oriented techniques when developing software:

- ☐ Always
- ☐ Sometimes
- ☐ Occasionally
- ☐ Never

Finally, for each of the following statements, would you strongly agree, agree, have no opinion, disagree or strongly disagree?

	Strongly agree	Agree	No opinion	Disagree	Disagree strongly
18. Component orientation is easily adopted into a development process					
19. Component technologies can be adopted independently of wider organisational consideration					
20. Project management is unaffected by component technologies					
21. Component orientation makes software reuse easy					
22. Component orientation should focus upon software reuse					
23. Using component technologies is straightforward					
24. A component oriented approach encourages design					
25. Component based development makes system deployment easier					
26. Component based development makes system maintenance easier					

If you are interested in the results of this survey please include your email address below:

E. Survey Respondents

Survey respondents were encouraged to include their name, position and organisation with a survey response. While some preferred to remain anonymous, those who did include respondent information are listed below.

Patrick Gleeson, R & D Developer, KPN Research, Holland

Ingo Stengel, Telecommunication Engineer, Univ. of Applied Sciences Darmstadt

Holger D. Hofmann, Software consultant, ABB Group, Germany

Hermann Kurth, software engineer and project leader, Mannesmann Mobilfunk

Ralf Kretzschmar-Auer, Chief Architect, dv/d systempartner

Dr Ulrich Eisenecker, Professor, University of Applied Sciences Kaiserslautern

Ralf Reussner, Phd-Student, Univ. Karlsruhe

Paul Dowland, Research Student, University of Plymouth

Mike Evans, R & D consultant, Glowebs

Andrew Watson, Technical Director, OMG

Charles Jursch, CTO, Patotech Software, Inc.

Jeff Watson, Sr. PC Developer / Webmaster,

Alex Goodstein, Director / IT Consultant, Linkform Computing Ltd.

Huseyin Caglayan, Developer, IT Innovation Centre

Scott Butler, President, Tango Enterprises, Inc.

Johnny Baron, Software Leader, Oramir

Frederic Gos, Advanced Software Engineer, Novo Nordisk IT A/S

Chris Sells, Director of Software Engineering, DevelopMentor

Darayush Mistry, Senior Consultant, Siebel

Roger Wolter, Program Manager, Microsoft

Stephen McKeown, Programmer, DTSC

Kirill M Katsnelson, Sr. Software Architect, Datamax Technologies, Inc.

Dean Olynyk, Technical Architect, black box consulting Inc.

Alexander Jerusalem, CTO, Vienna Knowledge Net

Michael Rees, Associate Professor in Computer Science, Bond University, Australia

Tim Kemp, Development Manager, Netscient Ltd.

Chris Sanders, R & D Director, Netscient Ltd.

Bill Slater, Software Developer, WR Engineering

Nicholas Moss, Business Consultant, DSTC

Tim Korson, Senior Partner, Software Architects

Jamie Cornes, Systems Engineer, DSTC

Alexey A. Ryaboshapko, Lead Programmer, Argussoft Co

Mike Siddall, Lead Programmer, Genesis Development

Albert Pi, Senior Programmer, PCI Inc.

G. Papers and Presentations

Papers

"A Distributed Component Framework for Integrated Network and Systems Management", Martin Knahl, Andy Phippen, Holger Hofmann. Information Management and Computer Security, Volume 7, Number 5.

"Online Distance Learning: Expectations, Requirements and Barriers", Steven Furnell, Mike Evans, Andy Phippen, Mosa Ali Abu-Rgheff. Virtual University Journal, 1999.

"A Hyper Graphics Markup Language for Optimising WWW Access in Wireless Networks", Paul Reynolds, Steven Furnell, Michael Evans and Andy Phippen. Euromedia 1999, Munich, Germany April 1999.

"Strategies for Content Migration on the World Wide Web", M.P. Evans, A.D. Phippen, G. Mueller, S.M. Furnell, P.W. Sanders, P.L.Reynolds. Internet Research Volume 9 Number 1. 1999.

"Content Migration on the World Wide Web", M.P. Evans, A.D. Phippen, G. Mueller, S.M. Furnell, P.W. Sanders, P.L.Reynolds. Published in Proceedings of the International Network Conference 1998. University of Plymouth. 1998.

"Mobility Considerations for Integrated Telecommunications Service Environments", M.P.Evans, S.M.Furnell, A.D.Phippen, P.L.Reynolds. Published in the Proceedings of the Sixth IEE Conference on Telecommunications. IEE Conference Publication No. 451. ISBN 0-85296-700. 1998.

"A Software Platform for the Integration of a Mobile Client to Intranet Service", Andy Phippen, Chris Hindle, Steven Furnell. Published in the Proceedings of Euromedia '98. Society for Computer Simulation. ISBN 1-56555-140-0. 1998.

"Network Resource Adaptation in the DOLMEN Service Machine", M.P.Evans, K.T.Kettunen, G.K.Blackwell, S.M.Furnell, A.D.Phippen, S.Hope and P.L.Reynolds.Published in Intelligence in Services and Networks: Technology for Co-operative Competition, Mullery et al. (eds.), Springer, 1997.

"Resource Adaptation in the TINA Service Environment", M.P.Evans, A.D.Phippen, S.M.Furnell, P.L.Reynolds. Published in the Proceedings of the Fourth Communication Networks Symposium. Manchester Metropolitan University. 1997.

As well as the papers above, numerous internal publications were written for the DOLMEN project. Additionally, contributions were made to three DOLMEN public deliverables.

Presentations

“Adopting and Using Component Technologies – an Organisational Learning Approach”, invited presentation Mannheim, Germany, summer 2001.

“Experiences with CORBA and Distributed Systems”, invited presentation to technical management at Wandell and Golterman, January 1999.

“Component Architectures and their Impact upon Software Development”, presented to the Distributed Applications Research Group, Fachoshule Darmstadt, Germany, June 1998.