UNIVERSITY
of
GLASGOW

Department of Computing Science

— PH.D. THESIS —

A Generic Feedback Mechanism for Component-Based Systems

by

Karen Vera Renaud

Submitted for the degree of
Doctor of Philosophy
University of Glasgow

June 2000

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Abstract

Computers have been integrated into all spheres and occupations and the need for users to easily understand how to use each computer application has become paramount. The end-user should not be expected to decipher cryptic messages or to understand the inner functioning of the computer itself. With computer-users spanning all walks of life, there is a need for a change in the mind-set of software developers in making their product more user-friendly.

In addition, software systems of the future will increasingly be built from independent encapsulated software components and will often be distributed over various sites. This new paradigm brings a new realm of complexity for the end-user, especially with respect to the increased possibility of failure, so that in addition to the non-trivial task of interpreting the general functioning of an application, the user will be expected to deal with the results of perplexing errors too. The nature of component-based systems makes the provision of support for handling errors far more difficult due to the independent and diffuse nature of the creators of the individual parts making up these systems.

Other factors with respect to application use also need to be addressed. For example, it is a rare user who is able to spend 100% of his or her time concentrating on interaction with the computer, without distractions of some sort interrupting. It is even rarer to find an application which is not prone to occasionally unintelligible error messages or breakdowns. Few applications are designed with these realities in mind and when problems do occur, or users are interrupted, they often find it difficult to recover and to resume their primary task. It is also difficult for applications to tailor the provided feedback according to the specific needs of different end-users or the differing roles within which they function.

This dissertation will highlight the role of feedback in increasing the interpretability of an application and in alleviating the effects of interruptions, errors and breakdowns. Rather than expecting feedback to be provided by programmers, this dissertation will argue that feedback can be enhanced in a distributed component-based system by separating the feedback concern from the basic functional concern of the application and executing the application within a generic feedback enhancing framework. The feedback concept is examined in depth and the role of feedback in enhancing understanding of applications, and in alleviating the effects of disturbances in our working day, is explored. The concept of a generic framework for enhancing feedback has been developed and a prototype implemented. The design and implementation of this prototype are described, as is the evaluation of the feedback thus produced.

Non nos sed Deus

Cardross Village Motto

To the men in my life —

my husband, Leon
my sons, Gareth, Ashley & Keagan
and my father, Philip Howard

Acknowledgements

I was assisted and supported by many people throughout the production of this dissertation, and during the course of the accompanying research. I would like to single some out for special thanks:

- ⋆ My supervisor, Richard Cooper, for his support, motivation, astute comments and insights, unfailing patience, and for his unswerving faith in me.
- ⋆ My husband, Leon, for encouraging me in this endeavour, and for his love and understanding. Without his support I would not have embarked on this journey of discovery.
- ⋆ My sons, Gareth, Ashley and Keagan, for putting up with the side-effects of my efforts to get a PhD so graciously and for being such outstanding young men.
- ⋆ Huw Evans, for his friendship, wicked sense of humour, and painstaking attempts at refining my prose.
- ⋆ Malcolm Atkinson, for his support and for constructive comments on drafts of this dissertation.
- ⋆ Ela Hunt, Phil Gray, Ray Welland, Stuart Blair and Susan Spence, for spending time discussing my work with me and for their extremely helpful comments.
- ⋆ Rosemary McLeish, for her friendship and for proof-reading this dissertation.
- ⋆ My family in South Africa: Mom & Dad, Basil & Leonie, Wendy, Felicity, Ken & Joan, Lynn and Bernice. They supported us from afar, and we could not have borne the separation without their love and assistance (and regular emails).
- ⋆ Vera and Ian McCulloch, our adopted Scottish "family". They gathered us under their wing and made us feel at home. *Dankie, en Weereens Dankie.*
- ⋆ Neill Bogie, for being brave enough to test my prototype.
- ⋆ The Association of Commonwealth Universities, and the Foundation for Research and Development in South Africa, who provided the funding for this research.
- ⋆ The University of South Africa, for allowing me an extended period of absence to undertake the research necessary to do this degree, and for financial assistance too. Thanks to the Head of Department, Paula Kotzé, for her continued support.
- ⋆ BEA Systems, who donated the use of their Tengah Server for the duration of this research.

If it were done when 'tis done, then 'twere well it were done quickly.
William Shakespeare. Macbeth. Act 1 Scene 7

Contents

I Prologue 1

1 Introduction 2
1.1 Thesis Statement . . . 2
1.2 The Shortfall in Application Feedback . . . 2
1.3 Feedback in Component-Based Systems . . . 4
1.4 Potential Solutions . . . 5
1.5 Road Map . . . 6

II Summary of Background Material 7

2 Software Components 8
2.1 Why Components, and What are They? . . . 8
2.1.1 What are Components? . . . 10
2.1.2 How are components different from objects? . . . 13
2.2 The Component Runtime Environment . . . 15
2.2.1 From Two-tier to Three-Tier Architectures . . . 15
2.2.2 The Middle, or Business-Logic, Tier . . . 18
2.2.3 From a Component-Framework Middle Tier to Component-Oriented Middleware . . . 19
2.2.4 Moving to N Tiers — The Impact of the World Wide Web . . . 22
2.2.5 Summary . . . 24
2.3 The Evolution of Components . . . 24
2.3.1 Components Embedded within a Process . . . 25
2.3.2 Components in Different Processes . . . 27
2.3.3 Components on Different Machines . . . 27
2.4 Prominent Component Models . . . 29
2.4.1 The OMG's Component Model . . . 31
2.4.1.1 Architecture . . . 31
2.4.1.2 Middle-Tier Architecture . . . 33
2.4.1.3 Example . . . 33

2.4.2 Sun's Component Model . . . 34
2.4.2.1 Architecture . . . 34
2.4.2.2 Middle-Tier Architecture . . . 35
2.4.2.3 Example . . . 36
2.4.3 Microsoft's Component Model . . . 36
2.4.3.1 Architecture . . . 37
2.4.3.2 Middle-Tier Architecture . . . 39
2.4.3.3 Example . . . 39
2.4.4 Summary . . . 40
2.5 Component-Based Development . . . 42
2.5.1 A Different Approach . . . 43
2.5.2 Component Sources . . . 44
2.5.3 Benefits of Using Components . . . 45
2.5.4 Summary . . . 46
2.6 Review . . . 47
2.6.1 The good news about components . . . 47
2.6.2 Reasons for cautious acceptance of components . . . 48
2.7 Conclusion . . . 49

3 Quirks **51**
3.1 Introduction . . . 52
3.2 Analysis of Quirks . . . 53
3.3 Why Quirks are Important . . . 54
3.4 System Crashes and Breakdowns . . . 56
3.5 Human Error . . . 63
3.5.1 The Nature of Error . . . 63
3.5.2 Performance Levels and Likelihood of Errors . . . 65
3.5.3 Detecting Errors . . . 66
3.5.4 Enabling User Understanding of Error . . . 68
3.5.5 Recovering from Error . . . 69
3.5.6 Summary . . . 73
3.6 Interruptions . . . 73
3.6.1 Nature of Interruptions . . . 74
3.6.2 The Composition of an Interrupt . . . 77
3.6.3 Dealing with Interruptions . . . 78
3.6.4 Summary . . . 81
3.7 Summary . . . 81

4 Feedback **82**
4.1 Introduction . . . 82
4.2 Purpose of Feedback . . . 85
4.3 Why give Feedback? . . . 86
4.4 When must Feedback be Given? . . . 88
4.5 What is Good Feedback? . . . 89
4.5.1 Examples of Inadequate or Bad Feedback . . . 91
4.5.2 List of Desirable Feedback Features . . . 93
4.5.3 Provisos . . . 94
4.5.4 Differing User Roles . . . 96
4.6 Feedback Format . . . 96
4.6.1 Textual versus Graphical Feedback . . . 96
4.6.2 What Does Visualisation Do? . . . 98
4.6.3 Restrictions . . . 98
4.7 Feedback for Quirks . . . 99
4.7.1 Breakdowns . . . 100
4.7.2 Human Error . . . 100
4.7.3 Interruptions . . . 100
4.7.4 Conclusion . . . 101
4.8 Summary . . . 102

III Addressing Feedback Needs in Component-Based Systems **103**

5 Problem Description and Proposed Solution **104**
5.1 The Problem . . . 105
5.1.1 Traditional Ways of Providing Feedback . . . 105
5.1.1.1 Guidelines for Programmers . . . 105
5.1.1.2 Comprehensive Online Manuals . . . 106
5.1.1.3 A Feedback Application Programmer Interface . . . 106
5.1.1.4 Summary . . . 106
5.1.2 Why Feedback Provision is (Even More) Difficult in Component-Based Systems . . . 106
5.1.3 Why Error Recovery is (Even More) Difficult in Component-Based Systems . . . 107
5.1.4 Conclusion . . . 111
5.2 The Proposed Solution . . . 111
5.3 First Mechanism — Separation of Concerns . . . 113
5.3.1 Separate Specification of Concerns . . . 113
5.3.2 Orthogonality of Concerns . . . 115
5.3.3 Summary . . . 115

5.4 Second Mechanism — Application Tracking 116
5.4.1 First Perspective — User-Interface Tracking 116
5.4.2 Second Perspective — System-Level Monitoring 117
5.4.3 First Approach — Invasive Tracking 118
5.4.4 Second Approach — Non-invasive Tracking 119
5.4.5 Summary 120
5.5 Third Mechanism — The Visualisation 120
5.5.1 Visualisation of User Interaction with an Application 120
5.5.2 Visualising Execution of Software 121
5.5.3 Visualising Dialogue 121
5.5.4 Visualising serial periodic data 121
5.5.5 Interacting with the Visualisation 122
5.5.6 Conclusion 122
5.6 Consolidation 122
5.6.1 Benefits of the Proposed Approach 123
5.6.2 Limitations of the Proposed Approach 124
5.6.3 Summary 124

IV HERCULE — Design and Implementation 125

6 HERCULE's Design 126
6.1 Design Philosophy 127
6.1.1 Design Principles 127
6.1.2 Accessing HERCULE 129
6.1.3 Required Application Features 130
6.1.4 And Thus... 132
6.2 Facilitating HERCULE's Observation Function 132
6.2.1 The "Minimal Impact Proxy" Design Pattern 134
6.2.2 The User-Interface Proxy 136
6.2.3 The Component Proxies 140
6.3 Facilitating HERCULE's Explanatory Function 142
6.4 HERCULE's Architecture 143
6.4.1 Communication modules 143
6.4.2 Controller 144
6.4.3 The Window Manager 145
6.4.4 The Server Proxy Manager 149
6.4.5 The Display Controller 152
6.4.6 Hercule Components 152
6.5 Application Activity Visualisation 153
6.5.1 How Should the Application Activity Visualisation be Provided? . . . 154

6.5.2 Visual Representation . . . 156
6.5.3 Layout . . . 157
6.5.4 Customisation . . . 159
6.6 Conclusion . . . 161

7 Implementation **163**
7.1 Prototype Application . . . 164
7.2 Observing User-Interface Activity . . . 165
7.2.1 Java Platform-Independent User-Interface Mechanism . . . 166
7.2.2 Inserting the Proxy . . . 167
7.2.3 The User-Interface Proxy . . . 168
7.2.4 Watching User Activity . . . 170
7.2.5 Maintaining and using the internal image of the GUI . . . 172
7.3 Observing Server Communication . . . 173
7.3.1 The Enterprise Java Beans Component Model . . . 173
7.3.2 Using Proxies to Intercept Communication . . . 176
7.3.2.1 Inserting the Proxies . . . 176
7.3.2.2 Sending the reports to HERCULE . . . 179
7.3.3 Using the reports generated by the proxies . . . 179
7.4 The Descriptor Tool and Proxy Generator . . . 179
7.4.1 The Descriptor Tool . . . 180
7.4.2 The Proxy Generator . . . 182
7.5 The Runtime Feedback Tool . . . 183
7.6 Application Activity Visualisation . . . 184
7.6.1 Characteristics of Visualisation . . . 184
7.6.2 Interactivity of the Display . . . 185
7.6.3 Extensibility of the Display . . . 187
7.7 Conclusion . . . 191

V Epilogue **192**

8 Evaluation **193**
8.1 Current Approaches to Evaluation of Tools . . . 195
8.2 Preliminary Evaluation Results . . . 196
8.2.1 User Needs . . . 197
8.2.1.1 Feedback . . . 199
8.2.1.2 Quirks . . . 200
8.2.2 Component-Based System Development and Maintenance . . . 203
8.2.2.1 Programmer Needs . . . 203
8.2.2.2 Programmer Experience with HERCULE . . . 204

8.2.3 Performance Impact 205
8.3 Conclusion 206

9 Conclusion **207**
9.1 Reiteration of Thesis Statement 207
9.2 Summary of Research 207
9.3 Thesis Contribution 209
9.4 Future Research 210

VI Appendices and Bibliography **212**

A Glossary **213**

B Minimal Impact Proxy Design Pattern Code **216**

part I

Prologue

What we call the beginning is often the end
And to make an end is to make a beginning
The end is where we start from.

T S Eliot. 1944

I have striven not to laugh at human actions, not to weep at them, nor to hate them, but to understand them.

Baruch Spinoza

Tractatus Politicus (1677) ch.1, sect 4

chapter 1

Introduction

1.1 Thesis Statement

I submit that feedback can be enhanced in a distributed component-based system by executing the application within a generic feedback enhancing framework. I further submit this supports the user: firstly in understanding the application, secondly in recovering from errors, and thirdly in rebuilding mental context after interruptions. The framework standardises feedback provision, simplifies application code, allows continuous post-implementation refinement of explanatory messages and promotes reuse.

1.2 The Shortfall in Application Feedback

The feedback provided by applications in general use is typically patchy — often inadequate and sometimes even misleading. Users often have great difficulty in ascertaining exactly what the application is doing with their inputs and consequently struggle to build up an internal model of how they should interact with the application.

The immediacy of the reactions of computers, combined with the fact that such reactions are not random but considered (having been designed by a human programmer), lead people to consider the computer to be a purposeful social object [Suc87]. Therefore the computer application can be thought of as fulfilling the same role as a conversational participant

[PQS96].

Participants in a human-to-human conversation do not merely take turns, but in many ways collaborate in the conversation. The speaker expects a level of feedback which is essential in gauging the listeners' reaction to what is being said, their understanding of the current subject, their opinions, emotions and much more. This could be referred to as indicating the listeners' "state of mind" and feedback can be considered to play a crucial role in assisting the speaker in interpreting this state. The speaker's interpretation of this state will play an important role in steering the conversation in one direction or another. During the discourse informal "rules" of conversation between two people are developed. In the same way, application feedback assists the user in understanding the interaction "rules" of the application.

In gaining an understanding of application interaction rules, the user often gets little assistance, since applications frequently do not explain themselves appropriately. Inextricably bound up with this is the related difficulty of perceiving the relevant aspects of the current state of the application. The computer's functioning and internal state are completely imperceptible, making its true nature even more of a mystery than it should be.

What we need to facilitate better communication between the application program and the end-user is, firstly, a way for the application to explain the interaction rules to the end-user and, secondly, a method of making relevant application state more available and perceptible. These two requirements can be termed the "interpretability" problem.

Full interpretability is difficult to achieve, since there is a fundamental mismatch and perennial misunderstanding between end-users and application programmers. This mismatch is exacerbated by the fact that application programmers produce applications which must communicate with a person about whom the programmer can make very few informed assumptions. Economic realities make it infeasible to develop an entire application for a specific user and consequently applications are produced for a "generic" user. There is a tendency to generalise the application interface to satisfy all the needs of generic users, yet this generality makes it difficult for individual users, with vastly different levels of experience and individual requirements, to understand the application's rationale.

The feedback channel, which is so vital to human-to-human interaction, can be utilised to enhance the interpretability of the application by conveying relevant information about the application's expectations and understanding of, and response to, the user's instructions. Feedback is routinely used to indicate either a confirmatory response, or to give information about the *function* of some user-interface component — by means of colour changes, balloons, icons etc.

Feedback with respect to application state is less common and far more difficult to provide correctly. It is difficult for an application programmer to know which aspects of the current application state should be visualised to enhance interpretability, which could serve no purpose and which would be positively confusing. Most rare of all types of feedback is information which tells the user how the current state was achieved. This causes problems

since human discourse is incremental and conversants will typically refer to something they have previously said. Human-to-application interaction seldom fosters this type of referral, which could potentially be very well catered for by an enriched model of feedback.

Furthermore, application programmers are often unrealistic about the the user's working environment and seldom cater for the effects of events which will interfere with the use of the application. Such events can disrupt the straightforward execution of a task and interfere with a user's concentration. These events, which will be referred to as *quirks*, could be system breakdowns, various types of interruptions to application use or human errors. Applications often make no concession to the inevitability of quirks and seldom give assistance in rebuilding mental context afterwards or facilitate understanding of the cause in the case of an error.

1.3 Feedback in Component-Based Systems

Component-based three-tier systems are the latest paradigm shift in the software engineering industry. It is widely believed that future computer systems will be built from software components. Unfortunately, they present new problems and opportunities which cannot be ignored. Whereas interpretability is a very real problem in traditional monolithic applications, it becomes an even bigger problem in component-based applications due to the independent and disjointed nature of the programming activity which produces the individual components used to build the system, and also due to the "black-box" nature of said components. The distributed nature of these systems increases the probability of errors and breakdowns, once again reducing interpretability.

The developers of the different components used to build a component-based application will seldom communicate with one another. The application will generally be constructed from pre-developed components and the developer of the front-end application will merely be given interfaces to these components specifying the contractual responsibilities and functionality of the component.

The developers therefore cannot enhance the feedback provided by the component, since they have no control over the implementation details and have to accept the feedback provided by the component, whatever its quality. The developer will also have great difficulty in anticipating all the possible error situations which could arise from the use of a server component. The encapsulation principle which drives component-based development gives system engineers the flexibility to be able to change the implementation of a component during the lifetime of the system. This could precipitate a whole new range of errors, hitherto unsuspected, which will probably be reported to the user in all their technical verbosity, reducing the user's understanding of the system and perhaps necessitating intervention by specialists.

The background knowledge of the target user of a component-based application is harder to gauge than that of the the user of a monolithic system since the distributed nature of

the applications is likely to mean a wider range of users. These systems are designed to support many different styles of front-end and to be made available on the internet, whereas stand-alone local deployment was previously the norm.

1.4 Potential Solutions

Programming applications in component-based systems is no easy task [Jam99b]. The current approach to providing feedback is an expectation that the programmer will program this *in addition to* building code which copes with all the complexities of the distributed system. This approach appears to be flawed, as evinced by current standards of feedback which do not always meet the requirements. It is also not economically viable to meet reasonable standards from within each application. This approach also leads to inconsistent provision of feedback making it difficult for the user to find and assimilate feedback when having to use several applications.

Current approaches to enhancing the interpretability of the system rely heavily on either paper or online manuals. The benefits of this approach are limited since research has shown that users seldom consult manuals, preferring to familiarise themselves with an application by using it [CR87].

An alternative approach, described in this dissertation, is that feedback be provided by a generic feature, produced *independently* of the application implementation. This approach necessitates treating the provision of feedback as a *separate concern.* This well-established technique has been successfully applied in separating several non-functional characteristics from the main concern of application programs, but has hitherto not been applied to the provision of feedback. Separating feedback provision from the application makes things easier for the programmer and provides a mechanism for augmenting the feedback provided by the application itself.

There are many approaches to achieving separation of concerns [HL95]. One approach, application tracking, requires the least effort from the programmer and was thus the approach chosen. It is also the least invasive way of achieving the required separation of concerns. Application tracking is widely used for many purposes, but once again has not hitherto been used to augment application feedback.

A prototype implementation of this proposal has been implemented, in order to test the viability of the scheme. This prototype has been evaluated in terms of the original feedback needs identified at the outset of the research.

The success of the prototype application has shown that this means of augmenting application feedback can indeed be used and that it enriches the concept of feedback in such a way that it can enhance the interpretability of a component-based application.

1.5 Road Map

The dissertation has been divided up into different sections:

- Part I contains this introduction.
- Part II provides the background literature in component-based systems, quirks and feedback. Chapter 2 provides an overview of component models, component-based systems and component-based development. Chapter 3 explores the nature of quirks — those events which interfere with our straightforward use of applications. Chapter 4 examines the nature and format of feedback, with attention being given to the role feedback can play in alleviating the negative effects of quirks.
- Part III describes the problem being addressed, proposes a solution, discusses the techniques used in the solution and enumerates the advantages and limitations of the proposed solution. This discussion constitutes Chapter 5.
- Part IV describes the design (Chapter 6) and implementation (Chapter 7) of the framework prototype which was developed in order to test the proposals made in Part III.
- Part V evaluates the prototype (Chapter 8), concludes, summarises the contributions of this dissertation and discusses future work (Chapter 9).
- Part VI contains the appendices and bibliography.

part II

Summary of Background Material

Read, every day, something no one else is reading.
Think, every day, something no one else is thinking.
Do, every day, something no one else would be silly enough to do.
It is bad for the mind to continually be part of unanimity.

Christopher Morley

Pooh began to feel a little more comfortable, because when you are a Bear of Very Little Brain, and you Think of Things, you find sometimes that a Thing which seemed very Thingish inside you is quite different when it gets out into the open and has other people looking at it.

A.A Milne

The House at Pooh Corner. (1928) ch.6

chapter 2

Software Components

This thesis proposes a generic feedback mechanism suitable for applications built out of components. Therefore this chapter will introduce software component concepts, since these form an integral part of the research discussed in this dissertation. Section 2.1 describes software components. A typical component runtime infrastructure is discussed in Section 2.2. Section 2.3 discusses the evolution of components. Section 2.4 describes the three prominent component models, and Section 2.5 gives a brief overview of the process of component-based development. Section 2.6 reviews material presented in this chapter, while the final section concludes.

2.1 Why Components, and What are They?

Software components are by no means a new concept. They were first proposed by McIlroy back in 1968 [McI68]. He suggested that the software industry needed the mass production of software components which could be bought and assembled. Parnas [Par72] originally proposed a packaging technology which is not very different from the prevailing component technologies of today. However, it is only the progress of the past few years which can make

this dream a reality.

Components are the latest attempt by the information technology world to simplify the production and management of software systems, a task which is notoriously difficult to accomplish. Brooks [BF95] argues that this is due to four properties of software systems:

1. *Complexity* — software systems can exist in a large number of different states which have to be visualised, described and tested by a developer. This increases with scale because of the added complexity generated by objects interacting within the system.

2. *Conformity* — due to the nature of the human institutions and systems to which software systems must conform.

3. *Changeability* — no other kind of system is subject to as many pressures for change as a software system. This is because software is perceived to be easily changeable, and because user requirements often change with time.

4. *Invisibility* — Software is very difficult to visualise, making it very demanding for humans to understand its function comprehensively.

Object orientation was initially hailed as the solution to these problems [Cox90], but failed to address them *significantly* or to reduce software development time as much as was anticipated [O'C99]. Object orientation on its own has not made much of a difference to programmer productivity. Any C++ programmer will readily attest to this. The advent of Java *has* made a difference, since it hides a lot of the complexity inherent in other object-oriented languages. It would be more accurate to say that *pure* object-oriented languages have made a difference to programmer performance and productivity. However, even with Java, software development remains a complex task.

Software vendors are well known for jumping on the band-wagon and hailing the latest innovation as the solution to all problems. The aggressive marketing of CASE tools is an example of this. The important fact overlooked by, or perhaps conveniently ignored by, software development tool vendors in their quest for profits is that no single innovation can be the cure for all software development difficulties, just as no one medical breakthrough can be the answer to all health problems.

Some have hailed the advent of components as being the "one best way" of developing software [SW98]. Others advise caution [Cha99c, O'C99]. It is important to bear in mind that computing is a relatively new science, and that the software development process needs to evolve significantly before we can feel we have arrived at a sufficient understanding of the process to claim that the one best way of developing software systems has been found.

At this stage, each new discovery is a step towards better software life-cycle methodologies. Object orientation has certainly made a significant contribution and is presently seen by many to be the best systems design approach. Object-oriented programming languages such as Java make programming much simpler. All the indicators point towards software components as the next step in this evolutionary process.

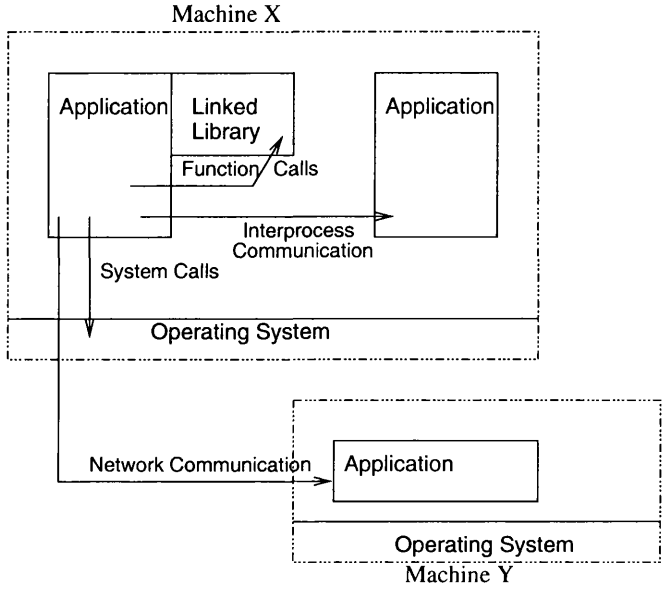

Figure 2.1: Different modes of Operation [Cha96]

The section heading posed the question: "Why should anyone use components?". One major reason is interoperability in the presence of increasingly heterogeneous contexts. The scenario presented in Figure 2.1 demonstrates different ways in which an application communicates with different types of entities. If a library is being used, it will be accessed via function calls. Operating system functions will be invoked by means of system calls. Communication with other applications is achieved by means of interprocess communication if the application is on the same machine, probably involving the use of the sockets mechanism. Communication is achieved by means of a remote procedure call if the application is on another machine. Communication with other applications, as well as with libraries, can usually only happen if both have been implemented using the same language.

Components provide the means for cross-platform and cross-application functionality. The component infrastructure offered by the prominent component models (to be discussed in later sections) enables a programmer to make use of the functionality within other applications, libraries and the operating system all in exactly the same way. Much of the complexity is hidden, and in addition, with two of the current component models, the implementation language is no longer an issue.

There are other benefits which make components attractive. The most important of these are their reusability and their appropriate size for reconstruction, marketing and assembly.

2.1.1 What are Components?

"Component", like "object", is an over-used word. It means many contrasting things to different people. Many different definitions exist for components, some of which are presented

here:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.[1]" [Szy98]
>
> "Software objects providing some type of known service, or specifications capable of creating such objects, that can be used in combination with other components to build systems via a well-defined interface." [Kar98]
>
> "An identifiable piece of software that describes and delivers a meaningful service that is only used via well-defined interfaces" [SW98].
>
> "A static abstraction with plugs" [NT95].
>
> "A way of packaging units or modules of software that makes them such that they could form some particular kind of plug standard." [WD98].
>
> "A component is a software module that publishes or registers its interfaces" [Har98].
>
> "A coherent package of software that can be developed independently and delivered as a unit, and that defines interfaces by which it can be composed with other components to provide and use services" [DW98].

Most of these definitions emphasise three important features: interfaces, a set of offered services, and reuse. Perhaps more helpful than a definition would be a list of the *required* and *desirable* properties of components. Components should, without a doubt [SW98, Cot98, Szy98, HG99]:

- be accessed only via interfaces, with each interface being a subset of the full contract the component has with clients. This implies the use of an interface definition language — both to enable the component user to discover properties of the component, and to enable the component developer to advertise services provided by the component;
- have explicit context dependencies — for example, if a component needs to access a relational database, the context dependencies would include information about the structure of the tables it requires. Location dependence is another example of a context dependency;
- be adequately documented — an essential part of the work described here;
- have a unique identity;

[1] This definition was first formulated at the 1996 European Conference on Object-Oriented Programming (ECOOP).

- be customisable;
- be a unit to be managed by a container — i.e. be more than just an executable binary. It must derive many of its properties from the container, and use facilities provided by the container. It must obey the rules of the container, and will have standard ways of sending events to the container [Pri99]. The services typically provided by a container include threading, transactions, security and persistence [Gut99].

Desirable characteristics include, but are not limited to:

- possession of a full description of possible exceptions which could be thrown, and explanations for these;
- the potential for the dynamic discovery of supported interfaces;
- minimum context dependencies;
- reusability.

The last two desirable characteristics will always conflict with each other. A component is most useful if it can perform its function without any restricting context dependencies, only relying on external services general enough to be provided by any component container. To achieve this, the component would have to have all required software bundled with it, but this type of over-inflated software produces exactly the type of problem that components were meant to solve.

If a component needs to make use of a secondary piece of software, it should rather request that service as a context dependency, so that the component only encloses software to execute its prime functionality. This makes the component eminently reusable since the prime functionality is probably specialised enough, and the component lightweight enough, to be used in other contexts as well. However, this reliance on external services makes the component more difficult to house because of the increase in context dependencies [Szy98].

For example, consider a desktop button, which is an eminently reusable object. Apart from the obvious requirement of the operating system, it requires the existence of a container within which it will be displayed. It will expect to inherit some of its properties from its container. The button will probably derive its background colour from the container, for example. Although it allows other components to register an interest in events executed on it, it requires the existence of an event-propagating mechanism which will inform it of user actions. The button can change its appearance and be tailored for any number of purposes in the user interface of most applications.

On the other hand, consider a desktop calculator component, which encloses all software and has few environmental requirements. Whenever it is used, it will have the same purpose — that of being a calculator. One cannot tailor it to other purposes, and though it is very useful, it does not fulfill the requirements of *reusability.*

This section has given details of characteristics that can be expected from software components. The following section will shed some light on how components are different from objects.

2.1.2 How are components different from objects?

Thus far the characteristics of components have been discussed. Some have criticised the component model as simply being the object model rephrased [O'C99]. Certainly components have many features of traditional objects, so perhaps the best way to start this discussion is by looking at the accepted notion of an object.

In the early days of object orientation people were confused about the meaning of objects too, and it was only after some time that the key concepts of object technology were distilled and universally accepted. The accepted object model tenets are [Tay99]:

1. *Objects* — executable software representations of real-world objects.
2. *Messages* — a universal communication mechanism through which objects interact with one another.
3. *Classes* — templates for defining similar objects.

The key *mechanisms* of object technology are [Tay99]:

1. *Polymorphism* — the ability to implement the same message in different ways to suit different object types.
2. *Encapsulation* — the mechanism for packaging related data and procedures together with objects. The aim of this mechanism is that objects should function as a black box, hiding the information and mechanisms for operating on the enclosed information.
3. *Inheritance* — a specialisation mechanism whereby one class can make use of information and messages defined within a generalised class.

The object-oriented community has had difficulty with the latter two mechanisms. Encapsulation was not equally well-supported by all object-oriented languages[2], and most languages' understanding of encapsulation did not extend to allowing applications developed using other languages to use their objects.

The relative desirability and particular nature of inheritance caused a great deal of debate in academic circles [Szy98]. Some organisations, such as Microsoft, argued that multiple implementation inheritance was a recipe for disaster. C++ allows multiple implementation inheritance, so that the programmer's life is made extremely difficult by unexpected side-effects of such inheritance. Implementation inheritance can also be regarded as a violation of encapsulation.

[2]The `friend` function in C++ is a direct violation of the spirit of encapsulation, and Java allows `public` variables, which can be manipulated by other objects

Quite apart from these problems is the generally acknowledged fact that objects themselves are too fine-grained to be deployed independently, because of their logical coupling with other objects [O'C99]. This limits the reusability of objects, and makes them difficult to use independently in a distributed environment. However, it is possible to identify a *group* of objects which collaborate with each other in providing some piece of functionality, and which form a type of unit, which would be more suitable for independent deployment. This collaboration can be deployed independently, as a separate component.

In attempting to distinguish objects and components, some key issues emerge — objects are fine-grained, while components are coarse-grained. Objects must be implemented in an object-oriented language, whereas components can be developed in any language. Objects do not always support encapsulation, but the very nature of components enforces encapsulation by the mandatory use of interfaces. Finally, objects are highly dependent entities, whereas components are designed to have a considerable measure of autonomy. Han [Han98] identifies some characteristics which, he argues, distinguish components from objects, with only components having the following characteristics:

1. *structural constraints* which will specify that certain compositions of attribute instances are not permitted;
2. *operational constraints* which specify permissible operation patterns;
3. *events* which can be fired by the component.
4. *multi-interfaces* which specify a number of roles the component can play.
5. non-functional properties such as security, performance, and reliability.

It is difficult to agree with this list. Java *objects* generate events, and inherit from multiple interfaces. Non-functional properties mentioned in point 5 are not generic component characteristics, but rather requirements enforced on behalf of the component by the container within which it is housed. Structural and operational constraints fall into the same category. The component has context dependencies, which incorporate all these requirements, although these are not properties of the component itself, since objects too can have these constraints and non-functional requirements. This list is therefore not useful in drawing a distinction between components and objects. While components *and* objects can be seen to share Han's properties, the true difference would appear to be that whereas objects have to implement the constraints themselves, components can expect many of the constraints to be applied by their container.

In summary, we can conclude that components *are* different from objects, mostly in terms of perspective. Object-orientation can be considered to be an *implementation technology*, while component technology is about *packaging and distribution.* Whereas the term "object" implies use of a specific type of implementation language, the term "component" should imply a unit of deployment providing a specific functionality.

Before continuing to the next section, which will discuss the component runtime environment, it is necessary, in the interests of clarity, to define the term component.

> *A* **component** *is a coherent, opaque, unit of software, accessible only via one or more interfaces cooperatively defining the full contractual duty of the component, which is independently deployed in a container enforcing and supplying the contextual requirements of the component.*

2.2 The Component Runtime Environment

In the not too distant past, all applications were monolithic and ran on what users deemed to be "powerful" and expensive mainframe computers. Users connected to these mainframe computers via "dumb" terminals. The mainframes were good at running such applications, and were not really tuned to reacting speedily to many requests from terminals. The monolithic applications were merely the automation of hand processing systems [BF97]. With the advent of the *Personal Computer* (PC), applications were simply moved from the mainframes to the PC. Moore's law[3] ensured that the PCs initially had no difficulty in keeping up with ever growing application resource demands.

However, the growth of the network and communications industry, the increasing demands of applications, and the difficulty of sharing data between users, changed the way people thought of applications, and the possibility of harnessing a powerful computer in the background to handle databases, for example, led to the advent, in the early 1970s, of client-server, or two-tier, systems. The following section will discuss the characteristics of these systems, and describe their metamorphosis into three-tier systems.

2.2.1 From Two-tier to Three-Tier Architectures

The two-tier architecture separates a distributed application into two collections of processes — clients which handle user interaction, and servers which manage resources. The form of inter-application interaction before the advent of these systems was facilitated by means of *Inter-Process Communication* (IPC). The IPC mechanism operates on a byte level, and the protocol for communication must be agreed upon by both participants, both of whom were probably implemented in the same language [Szy98]. In client-server systems, clients could, as an alternative to IPC, communicate with the server using a *Remote Procedure Call* (RPC) [BN84] mechanism. This mechanism places *stubs* on the client and server machines. When the client application makes a call to the client stub, it will marshal the parameters and send them to the server stub. The server stub receives the parameters, unmarshals them, and sends them to the server for processing. The client is unaware of this process and follows local calling conventions in using the procedure. The marshaling and unmarshaling process is responsible for conversions to different formats on different machines. RPC simplifies

[3] Moore's law implies that the power of computers doubles every 18 months.

all levels of communication (in-process, inter-process and inter-machine) by making their mechanism the same — the remote procedure call [Szy98].

There are two types of servers — *stateless* or *stateful* [Cor91]. The stateless server does not maintain any information about clients between calls. An example of this is a Web server. A stateful server "remembers" client information from one method invocation to the next. Stateless servers are more fault tolerant than stateful ones, since a client can simply keep resending a request till the server responds. The client of a stateful server needs to rebuild server context after a crash, and this could cause the client to fail. However, stateful servers operate in a well-understood programming paradigm, and are more efficient [Cor91].

In client-server systems, as shown in Figure 2.2, many clients use the same server on a request-reply basis. These architectures enable clients to have sophisticated user interfaces and data visualisation tools on their desktop computer, and share data with other clients by means of powerful database servers at the server level [BF97].

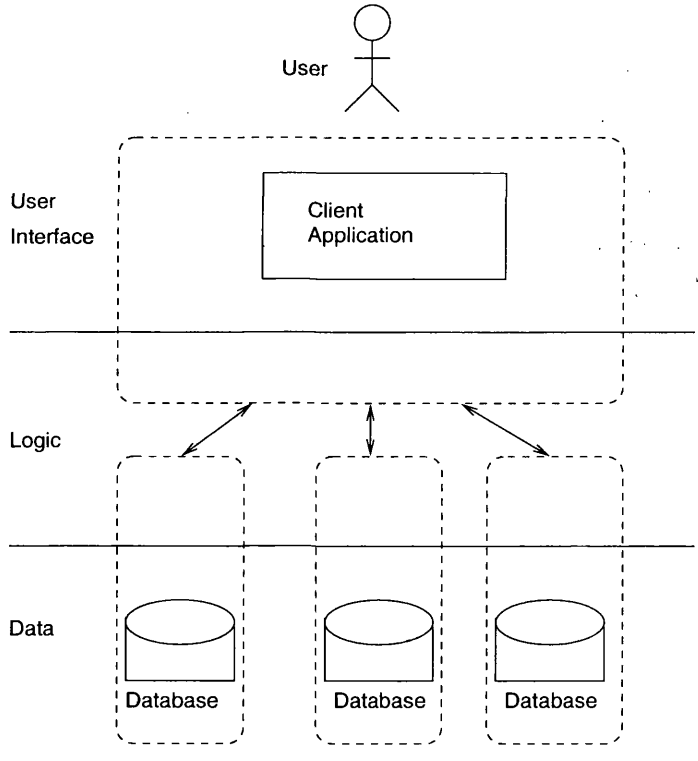

Figure 2.2: The Client Server 2-Tier Architecture

The client-server architecture was a great improvement on the previous monolithic systems, but had some serious shortcomings. There was a big question of where to put the application logic. If it is located in the client, the clients become "fat", and difficult to upgrade, and the application logic is too closely bound to the user interface code to reuse for another type of user interface. If a great deal of processing is to be done, it could adversely affect the performance of the client computer.

If the logic is located in the server, often a database server, it becomes tightly linked to the actual data source, and it is difficult to use data from other sources as well. It is also far too easy to overload the server, degrading performance and affecting response times to all clients. It is difficult to provide a reliable service due to the difficulty of load-balancing with this architecture.

Often the logic was split up between the client and the server, and it was very difficult to reuse any of the client code if the server type changed. If a different type of front-end were needed (for example, a touch-tone phone access front-end), a whole new application had to be written. It is also well-nigh impossible to integrate legacy systems into a conventional client-server system. In summary, client and server processes are too tightly coupled.

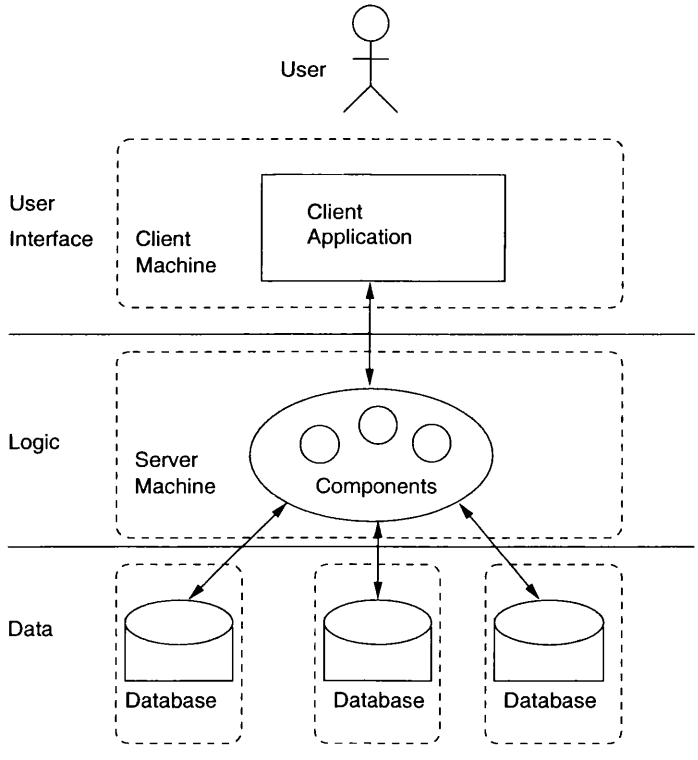

Figure 2.3: The 3-Tier Architecture

The three-tier architecture, shown in Figure 2.3, was developed to alleviate these problems, with the business logic being situated in a middle tier, between the client application and the data tier. The middle tier does not make assumptions about how a resource, such as a collection of data, is stored, but simply expects it to be provided by a lower tier. The user interface tier deals directly with the middle tier, and relies on this tier to interact with the lower tier and to control access to all shared resources. This new architecture makes it much simpler for different types of clients to share business logic and resources.

The middle tier could provide the same services to a desktop application, a browser, and a touch-type telephone, with only the user interface tier being specialised in each case. The

advantage of this approach is that the client becomes thinner, with most of the business logic being located in the middle tier. This minimises the cost of ownership of large numbers of client systems [Dol98].

Two of the three tiers in this model are well understood — the client application, and the lower (data) tier. However, the middle tier is relatively new, and will be described in the following section.

2.2.2 The Middle, or Business-Logic, Tier

The move to incorporate a middle tier was a logical response to the problems outlined in the previous section. However, when it came to planning the infrastructure, and the provision of the required middle-tier functionality, things were not as straightforward. It was necessary to share resources, such as data sources, and business logic, between different clients, and also to have a structure which was flexible enough to respond to changes in business rules without great difficulty.

The parallel development of independently deployable components, which were a viable alternative to finely-grained tightly-bound objects, made it possible for the business logic to be encapsulated in the form of middle-tier components. (The component evolution process is described in Section 2.3.)

The use of middle-tier components enables sharing between different types of applications, and the size and encapsulated nature of the components makes them easier to upgrade in a world of ever-changing business rules. In addition, instead of tying the business logic exclusively to one type of data source, components could be made flexible enough to link to any available data sources.

Since components are essentially evolved objects, the three-tier architecture requires the client to communicate with these components in an object-oriented manner — i.e. via method invocations. In this new architecture, therefore, the RPC mechanism is hidden from the programmer, and replaced by a remote method invocation protocol, because RPC does not directly support method invocations [Szy98][4]. A system of *proxies* is used to allow the client to invoke methods on a surrogate proxy object in exactly the same way as methods are invoked on local objects. The proxy object supports the same interface as the middle-tier component. The client program will request services from the middle-tier components by means of locally-based method invocations, and receive replies in the form of return values.

All components need to be housed within containers, which can provide essential services required by the components, such as, for example, life-cycle management, and administration functions. An infrastructure providing such services is called a *framework*[5].

[4]Method invocations require two things not provided by RPC: runtime inspection of the class of the receiving object to choose the method to be invoked; and the inclusion of a reference to the receiving object as a method parameter [Szy98].

[5]Lewandowski defines a framework as being *"a large design pattern capturing the essence of one specific kind of object system"* [Lew98]. Froehlich *et al.* define it as a reusable design and implementation of a

The object-oriented community originally used the constructs of object orientation to provide an object-oriented framework to house middle-tier components, giving birth to *component frameworks*[6]. The component framework provides support for the common functionality which is required by all components. Specific components can provide specific solutions, and make use of the framework to provide common features such as communication and life-cycle management. The framework imposes certain standards, and allows components to be "plugged in", to allow them to interact with groups of other components and the container itself in a standard way[Szy98].

Frameworks proved to be a suitable idea for taking care of some of the "wiring" problems of components, but had their limitations when providing for other special needs, which became clear as people gained experience in the use of three-tier component-based systems. The middle tier of an application could be servicing hundreds or thousands of concurrent users, and the types of problems to be dealt with could be [RS99]:

- How are client requests to be load balanced?
- How can system uptime be guaranteed in the face of system breakdowns and necessary administration?
- Is it possible to ensure that data in the lower tier remains consistent when being used by multiple users?
- How are client requests transferred to other machines in the case of a failure?
- How will clients be authenticated and authorised to perform secure operations?

The object-oriented community have had no history of dealing with these types of problems, and in order to solve them, they turned to the database industry. The following section describes the solution to this problem.

2.2.3 From a Component-Framework Middle Tier to Component-Oriented Middleware

Transaction Processing Monitors (TPMs), such as IBM's CICS [GR93], have vast experience in dealing with the issues not dealt with by component frameworks — in the context of database systems — and the obvious solution to dealing with these issues for middle-tier components is to give component frameworks special TPM capabilities. TPMs are very

system or subsystem [FHLS99]. They describe a framework as being implemented as a set of abstract classes which define the core functionality of the framework, together with concrete classes for specific applications. Froehlich *et al.* point out that frameworks are intended to provide a generic solution for a set of similar problems, with applications providing a specific solution for a specific problem.

[6] "A dedicated and focused architecture, usually a few key mechanisms, and a fixed set of policies for mechanisms at the component level" [Szy98] (p275).

good at maintaining and utilising scarce resources such as database connections, and at co-ordinating distributed transactions. This led to a natural marriage of component framework groups and TPM groups, in order to create a new product which would be able to handle components, be scalable enough to control many distributed transactions, and deal with the issues mentioned in the previous section [Ses98b].

This new infrastructure has been called by many different names, often denoting the specific vendor implementation. For example, Dolgicer [Dol98, Dol99] calls it an *Object Transaction Monitor* (OTM), and Sessions [Ses98a] calls it COMWare. The different names denote the same functionality, implemented in different ways — which will be explained further in Section 2.4. In this text, it will be referred to as *Component-Oriented Middleware.* This concept will not have an associated acronym, since the one which readily comes to mind has already been used by Microsoft to describe their specific component model.

> *The concept of a* **component model** *will, in this text, refer to the full standard encompassing the component definition, packaging, container architecture, component-oriented middleware and communication infrastructure.*

The component-oriented middleware infrastructure takes care of transaction management, component life-cycle management, supports component packaging and distribution, communication, scalability and security. The consequences of this are that:

- The physical location of the middle-tier components is unimportant — they are used as if local to the client.
- Middle-tier components can be duplicated on, or moved to, other servers to meet increased demand, and to help guarantee required uptime.
- Process and machine boundaries are more easily crossed [Pri99]. Platform idiosyncrasies have ceased to be relevant because a standard interaction mechanism — the remote method invocation — provides a standard way of accessing any component instance, anywhere.
- A middle-tier persistence service will ensure consistency of the underlying data sources in the presence of distributed transactions.
- Security will be handled at the middle tier, rather than at the client tier, decreasing the chance of unauthorised access.
- Best of all, the application programmer no longer has to be concerned about *systems* issues such as security or transaction boundaries.

There are two approaches to providing access to these, and other, component-oriented middleware services:

1. The first, followed by CORBA Versions 1 and 2, is to provide them by means of different *Application Programmer Interfaces* (APIs) — as illustrated in Figure 2.4. The client has to invoke the different services — such as transaction management —

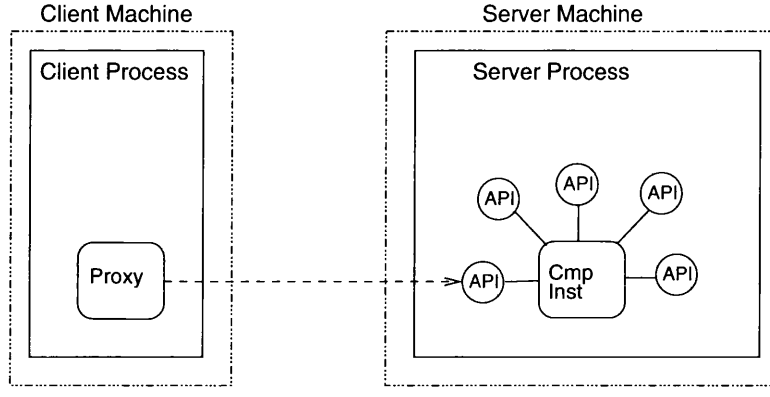

Figure 2.4: Middleware by API [Ses00]

in order to apply the service it requires to the component instance. This approach allows the client-application programmer to exercise control over these aspects of their component usage. CORBA was designed to be extensible, so that organisations could buy only those parts of the CORBA implementation they required, and services they would use.

However, middle-tier components only really become an asset if they ease the task of the application developer. While one can understand the motivation behind the extensibility and flexibility of the API approach, it does mean that the *application* programmer has a great deal of complexity to deal with which has little to do with the actual functionality of the program, and which should be the responsibility of a *systems* programmer. The alternative approach, interception, deals with this complexity on behalf of the programmer.

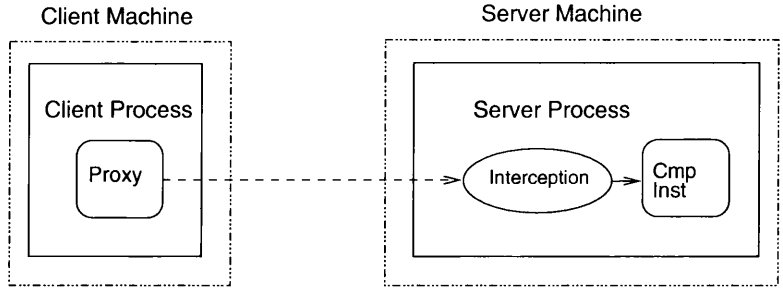

Figure 2.5: Middleware by Interception [Ses00]

2. The second approach to provide these services automatically is by means of intercep-

tion algorithms — as illustrated in Figure 2.5. The motivation behind a component runtime environment is that the component's needs are *declared* by setting component properties, instead of these issues being managed programmatically. The component-oriented middleware can fulfill these properties, without requiring any effort from the client. The result is rapid application development, and a shorter learning curve for the client-application programmer.

This approach has been followed by both Microsoft and Sun in their component models. The interception approach requires the server component to be housed within some sort of container, so that all client requests are intercepted and transaction boundaries can be enforced, life-cycle management can be achieved, and so on. The latest CORBA specification — Version 3 — also specifies the use of interception, rather than APIs, to invoke these services.

Sun provides the *Enterprise Java Beans* (EJB) component-oriented middleware specification. Microsoft provides COM+. The Object Management Group (OMG[7]) has accepted the advantages of the second approach, and has released its third version, which also applies the interception approach to providing component services.

2.2.4 Moving to N Tiers — The Impact of the World Wide Web

The advent of component-oriented middleware was soon followed by the tremendous success of the Web. Organisations began to see the need for Web-centric applications. There are two ways of making the component-oriented middleware web-wise:

One option is to add another tier to these systems, with the web server coming between the client and the component-oriented middleware. The *N-tier* architecture was thus born — with a tier for each major service provided in the middle tier, occurring between the client and the data tier. The new architecture is shown in Figure 2.6.

The other option is to make the component-oriented middleware itself Web-wise, leading to the *application server* concept. Application servers are often produced by professionals who have a lot of experience in the TPM world, such as IBM and BEA Systems[8]. The term "application server" will be used throughout this dissertation to refer to web-centric middle-tier component-oriented middleware.

Once again, there are many names for what is essentially the same concept. Taylor and Vaughan [TV99] point out that the term "application server" is often associated with the Java language. It is certainly true that the term "application server" is many things to many people, with as many problems in pinning down its true nature, as was experienced in pinning down the term "object" many years ago [Cha99b, VL99]. Indeed, IBM called their CICS servers, "application servers", long before the current middle-tier component-oriented flavour was attached to the term.

[7] `www.omg.com`

[8] `www.beasys.com`

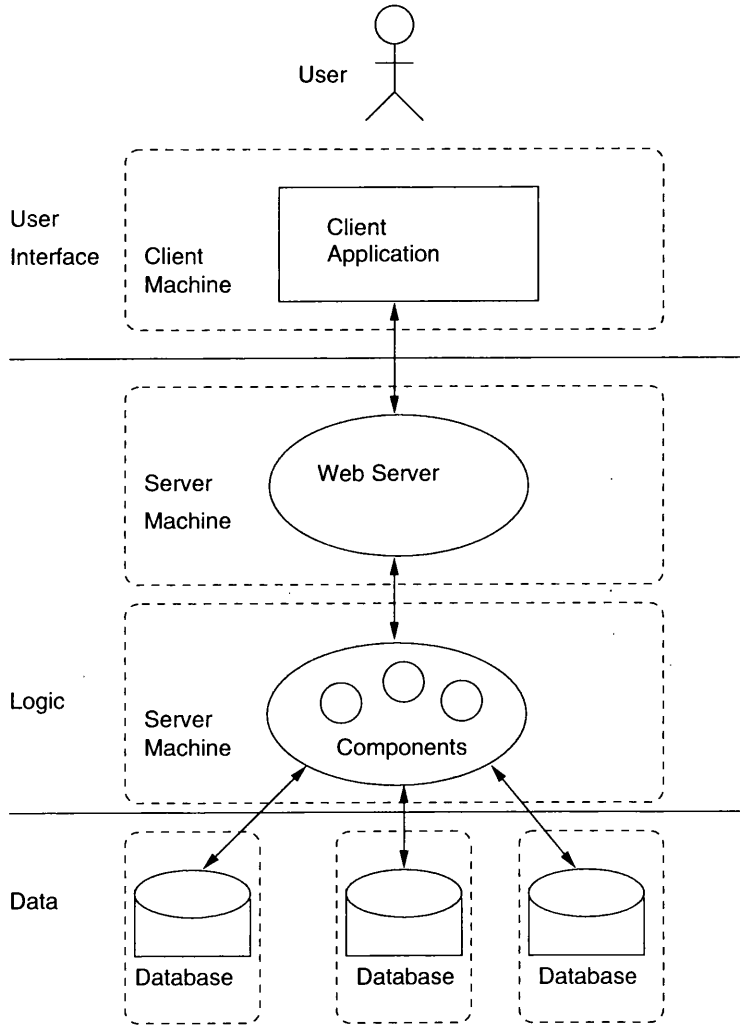

Figure 2.6: The N-Tier Architecture

Even though the exact nature of the term "application server" may be difficult for people to agree on, it is generally accepted as a term for a web-wise middle-tier structure, which is able to provide a reliable service and guarantee required availability. There is currently no argument about the fact that application servers play an increasingly important role in the development of enterprise applications [Mes98].

2.2.5 Summary

Previous application architectures — monoliths and client-server — failed to provide systems which were reliable, easily maintainable and flexible [BF97]. The three- and N-tier architectures recognise the fact that business rules are independent both of the user interface and the data source. These architectures offer organisations rich rewards because the middle tier, being specialised, can offer the following runtime services [Dol98]:

- load balancing, which might be delegated to the operating system;
- high availability and recovery;
- security;
- component management and monitoring;
- database connection management, shared cache and pooling;
- state/session management;
- result caching;
- location and service transparency.

This section has attempted to give an extremely condensed view of the vast field of distributed applications, the infrastructure that supports them, and the role that components play in these applications. The following section will take a closer look at the evolution of software components from the object to the middle-tier business-logic component.

2.3 The Evolution of Components

This section will give a synopsis of the advent and uptake of components by the computer software industry. While application architectures were moving from two-tier to *N*-tier systems, a parallel movement in the component world was taking place, moving components from specialised to generalised entities.

Section 2.1.1 described the differences between objects and components. Basically, objects were simply too fine-grained, not easily independently deployed, and had to be used from within the same language. The component concept dealt with these problems and

offered interoperability regardless of implementation language, and the chance of some disciplined reuse.

Specialised components have been used for quite some time in aircraft, power and automobile industries. However, the component industry, in the interests of interoperability, realised that they needed a standard way to access the services of components. The following decisions were made:

1. In the first place, components would exhibit the very valuable property of encapsulation. To enforce this, all components would be accessible only via an interface, thus separating the behaviour definition from the implementation. This also allows a measure of polymorphism to be applied, as well as enabling updating of components without interfering with client-application code.

2. Secondly, a standard mechanism for accessing the services offered by these components had to be decided upon, and since the above-mentioned interface mechanism was used to apply encapsulation, the standard mechanism would be method invocations.

3. Whereas there had been mechanisms for applications to interact prior to components, for example, by means of the socket mechanism, or messaging — as is done by IBM MQSeries (email for applications) — the popular component industry initially only used *synchronous method invocations* to interoperate. While asynchronous communication often achieves better performance than synchronous, such systems are very complex to design and debug [Szy98]. (The latest Microsoft component model, COM+, allows asynchronous communication, as does CORBA Version 3 — but these are later innovations.)

The advent and growth of the component industry can be traced from the initial embedding of components within a single process, followed by the use of components between different processes on the same machine, to the current use of components on different machines. The following subsections will trace these stages.

2.3.1 Components Embedded within a Process

The first *non-specialised* and popular component approach to be seen in general use was found in compound document models. One example of this is the *Object Linking and Embedding* (OLE) model from Microsoft. OLE documents embed or link to other subsidiary documents. When the user activates the subsidiary document, the necessary application is started, and the user interacts with it without leaving the context of the surrounding document. Compound documents are more user-centred, since they apply a document-centric approach rather than an application-centric approach [Szy98]. This means that the end-user does not have to be concerned with the particular application used to manipulate different parts of the document — text, clip art or diagrams, for example — but can merely be

concerned with the document itself, leaving these details to the application being used to create the document. Another compound document example is the Web, with components embedded in *Hypertext Markup Language* (HTML) pages which summon plug-ins to permit user interaction. The structure of an in-process component, which could represent an embedded component within a document, is shown in Figure 2.7. The small clear circles represent component interfaces. The filled circles represent references, or pointers, within the process.

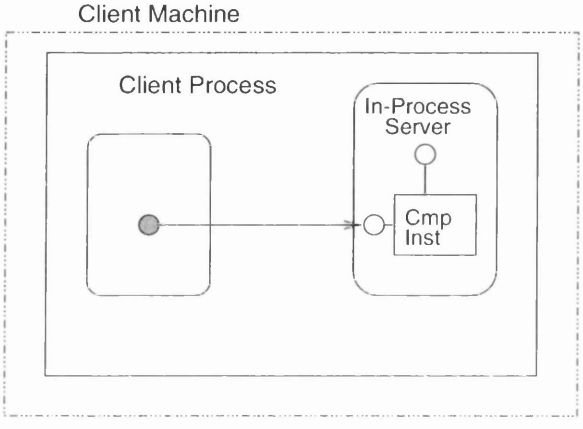

Figure 2.7: Microsoft view of Component within the Same Process [Cha96]

Soon after the advent of OLE, Visual Basic introduced 16-bit VBX controls, which were components originally intended to allow developers to create custom *Graphical User Interface* (GUI) objects for use within Visual Basic. However, developers soon started to use them to create other kinds of software components. They were then replaced by 32-bit OCX controls, and later by ActiveX — as the standard for Windows software components [Kar98].

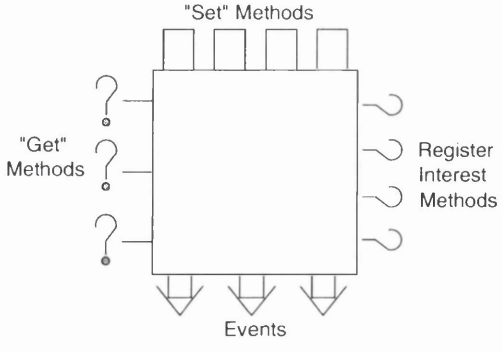

Figure 2.8: Structure of GUI Components

In October 1996 JavaSoft released JavaBeans [Eng97]. JavaBeans are similar to ActiveX controls because they are deployed at the desktop, but whereas ActiveX controls can be

developed in any of a number of languages, JavaBeans are developed in Java. The general structure of the GUI components is shown in Figure 2.8. Each of the components must have methods which can either set or query its properties. It also needs methods which allow the container process to register an interest in user actions on the GUI; and it generates events as a result of those user actions.

2.3.2 Components in Different Processes

Microsoft released their COM standard in the early nineties. This standard allowed interprocess use of components, as shown in Figure 2.9. While the components had all to be on the same machine, the COM standard provided a mechanism which uniquely identified components and their interfaces, and dynamically discovered the interfaces implemented by other components. A component loader sets up component interactions, and the interaction is relatively painless for the programmer using the components — especially when using a language such as Java.

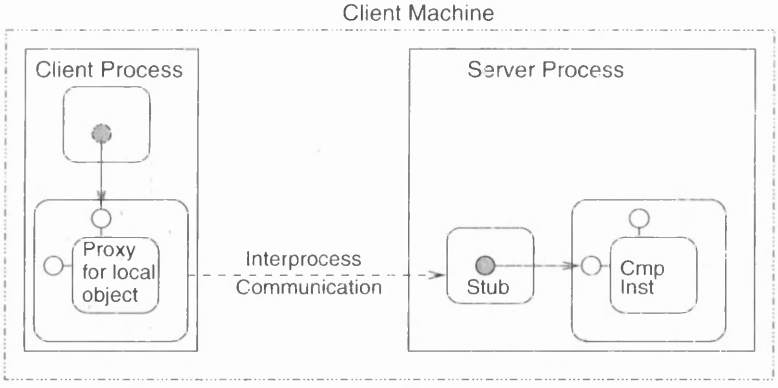

Figure 2.9: Components on the Same Machine [Cha96]

2.3.3 Components on Different Machines

While Microsoft was busy perfecting their component standard for desktop components, another parallel movement was working on the idea of distributed interoperability. This would allow components on different machines to make use of each other's functionality, as shown in Figure 2.10.

The *Advanced Networked Systems Architecture* (ANSA) originated in a project undertaken by a group of software development organisations within the United Kingdom Alvey Technology Programme in 1986 [ANS89]. ANSA wanted to provide an architecture for distributed systems which would be portable across different platforms, using different operating systems. They also worked towards providing a modular structure with maximum reuse of functionality. ANSA supported a range of distributed functions such as naming,

concurrency, and fault handling. Their basic premise was that architecture should adopt open standards wherever possible, and that this architecture should operate in such a way that the fact of distribution should be transparent to application programmers and users.

The *Object Management Group* (OMG), a software consortium founded in 1989, continued the work of ANSA, and started working towards a set of standards with the aim of promoting interoperability on all levels of an open market for 'objects' [Szy98]. They were working on specifications for a complete distributed object platform. Their focus was somewhat different from Microsoft's in that they were working on a specification which could be implemented by many different vendors, with the main purpose of allowing them to interoperate. Microsoft has always been quite open about the fact that in order to use their technology you have to use their operating system. They make no apologies for this, claiming that it makes their system more efficient, and that their approach is better than a set of standards which they claim to be unproven. A debate on merits of the relative positions is outwith the scope of this discussion.

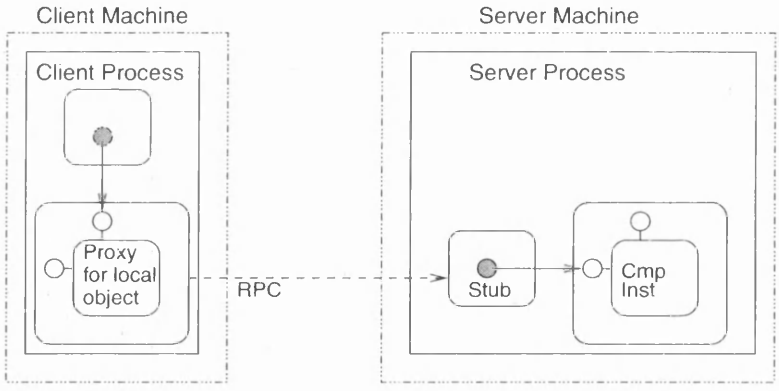

Figure 2.10: Components on Different Machines [Cha96]

After releasing their first standard in 1991, the OMG released their widely accepted and implemented CORBA Version 2 *de jure* standard in 1995. The latest standard, Version 3, was released in 1999. CORBA objects, while satisfying many of the requirements for components, are referred to simply as objects.

Microsoft quickly realised the potential for distributed components and released DCOM, which allowed the use of components between different machines. COM/DCOM soon became a standard for distributed Windows software components. DCOM treats the centralised option, where components are on the same machine, as a special case of the distributed option, as recommended by Stroud [Str93]. This approach allows the user to use components regardless of location.

Sun released the Java/RMI distributed object protocol, which could be used either on the same machine, or between different machines. This required both processes to have been written using Java, and made use of a naming mechanism called the `RMIRegistry` to allow

processes to locate components.

The use of distributed components in this way, which allows the user to use the remote component instance as if it were local, remains an illusion. In fact, this mode of operation invalidates a number of assumptions which could be made if the object were locally available. The assumptions are [ANS89]:

- *Failure:* more failure modes are possible for remote method invocation than for local method invocation. (More about this in Chapter 3.)
- *Binding:* configuration becomes a dynamic process, with bindings carried out at run-time.
- *Concurrency:* mechanisms are required to impose sequentiality when resources are shared by many clients.
- *Asynchronous communication:* required to support pipelining, and workflow processes.
- *Heterogeneity:* requires a common data representation for interactions between different systems.
- *Replication:* can provide availability and dependability.
- *Location independence:* necessary to enhance the availability and reliability of the system.

Local optimisations can be performed if the object in use is local, but local should be treated as a special case of distributed, not *vice versa* [Str93]. Such optimisations should never be implemented at a source code level, but rather at a compiler level [ANS89]. The following section will take a closer look at the three prominent component models.

2.4 Prominent Component Models

In the early days of components many organisations made use of the general *component* concept to develop their software. An example of this is OpenDoc, from Apple. OpenDoc is a component framework for visual components, with the components being called OpenDoc *parts*. OpenDoc did not conquer the market place, even though it was far ahead of the field, mainly because of marketing failure [Szy98]. Another example of a non-standard component approach is BlackBox, which is also a component framework for visual components. Neither of these component models have made a significant impact on the market.

The current component world sees three major players, JavaBeans/Enterprise Java Beans (EJB) from Sun, Common Object Request Broker Architecture (CORBA[9]) from

[9]While CORBA is often referred to as an *object* model, it has most of the features of component models, as discussed in the previous sections. The main reason why CORBA is often not considered to be a component model is because it is not based on the concept of an interceptive container architecture. This is a feature of the other two component models but it is not an essential component model feature.

the OMG, and COM+ from Microsoft. These standards are all still evolving, with CORBA having the oldest component standard (since 1991), and Microsoft having the most mature interception-based component-oriented middleware (since 1996). Each of the models will be discussed briefly in turn in the following subsections.

Each component model has different views about what a component is, how it should be implemented, how components should be located, how interfaces should be defined, and how components should communicate with one another and their environment. Although there are differences in the way that each of these works, there is a certain generic functionality which is required by all. Each has the following essential features [Ses00]:

1. *A component architecture.* The architectures focus on component packaging and interoperability. This includes:

   (a) the definition of an *interface definition language*, used by the designer to describe the component.

   (b) a *remote method invocation protocol.* This protocol specifies how the system will support remote method invocations on distributed objects.

   (c) features for *interoperating* with other component models, or the same one running on a different platform.

   (d) a *naming protocol* which enables the client application to search for a particular component.

   Client applications use this part of the component model to understand the component's features.

2. *A component runtime environment.* This is the container architecture, discussed fully in Section 2.2, which provides an environment for components. Components obey the rules of the container, and communicate with the container in a standard way. They also derive certain properties from their container [Pri99].

3. *Administration tools* — to manage the system and configure components.

4. *Interoperability service* — which allows the component runtime environment to communicate with external services. These could include [Pri99]:

   (a) *Persistence.* This service provides a uniform mechanism that allows transactions to be performed over one or more persistent data stores.

   (b) *Licencing.* This ensures that the users of components have paid to use it.

   (c) *Security.* This service ensures that the client is actually authorised to use the component, and controls privileges for different users.

   (d) *Messages.* This service supports asynchronous messaging.

(e) *Distributed garbage collection.* This service automatically deallocates distributed objects when they are no longer being used.

Built upon these similarities are differences in perspective, as will be evidenced in the discussions on the prominent component architectures in use today.

In order to illustrate the similarities and differences between these models an example will be introduced. Assume we have a server component called `CustomerComponent`, which holds the name and password of an organisation's clients. This component will interact with a relational database which stores information about customers. For the sake of simplicity, the `Customer` relation has only two attributes, `name` and `password`. The component supports two groups of methods. The first group consists of `getName()` and `getPassword()`, which are invoked to get information to validate a client. The second group has the methods `setName()` and `setPassword()` which are used to set up initial accounts for clients, or to change passwords.

2.4.1 The OMG's Component Model

The OMG is a software consortium, whose 800 current members have a shared goal of using integrated software systems [See98]. OMG members came together because they wanted to find a way for distributed object systems implemented in different languages on different platforms to be able to interact with each other. The first version of the CORBA specification was released in 1991 and the latest CORBA standard, Version 3, was released in 1999.

The OMG's main focus throughout their standards development has been that of interoperability. Many organisations have been involved in the development of their standards. This means that respected experts in industry and computing science have participated and that the possibility for a really good standard exists. However, their standards are complex, and often suffer from a "please everyone" syndrome. Big software firms routinely have their own unique ideas about how things ought to be done, and integrating strong opinions from different experts is no small task. The result is an often overly complex specification, with far more features than should be incorporated.

That said, CORBA has been widely accepted in industry today, and many implementations of the standard exist. Pritchard avers that the CORBA vendors are respected and that their products are perceived to be more appropriate for mission critical applications than COM [Pri99].

2.4.1.1 Architecture

The CORBA architecture is illustrated in Figure 2.11. The client and the CORBA object interact with the *Object Request Broker* (ORB) by using an *Interface Definition Language* (IDL) interface. Each CORBA object must have its interface specified in this IDL, and clients only ever see this interface — never the implementation. This separation guarantees

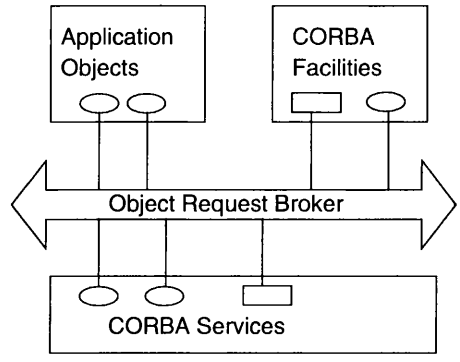

Figure 2.11: The CORBA Architecture [Sie98]

the substitutability of the object. The ORB builds on top of the network transport, using its own *Internet Inter-ORB Protocol* (IIOP) to communicate with other ORBs. This is illustrated in Figure 2.12.

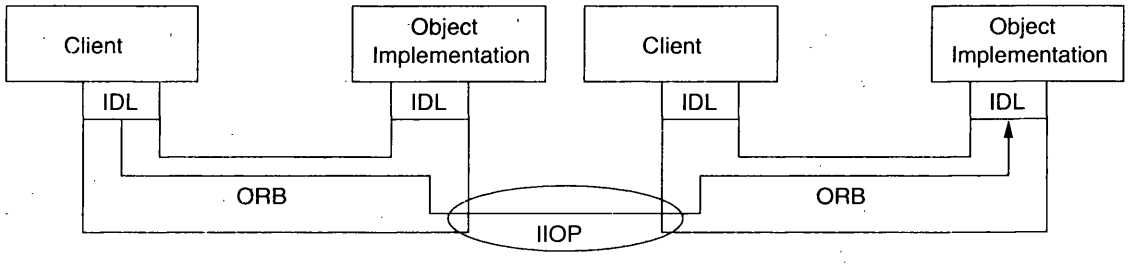

Figure 2.12: The InterORB Protocol

When a client makes a request, the request will be intercepted by the ORB, and passed to the target object. This happens for all objects, whether local or remote. The ORB is provided by means of library routines which manage both in-process and remote invocations transparently. CORBA supports distribution by having shared *Interface Repositories* (IR) which ensure that all ORBs on the network can gain access to all required IDLs.

CORBA makes use of common object service specifications (CORBAservices), and a set of common facility specifications (CORBAfacilities) to broaden its focus and provide specification for services to be used on top of the "wiring" provided by the ORB. The services provide things such as a naming service, transaction management, concurrency and other middleware requirements. The facilities which provide support for the enterprise by specifying standard objects for standard functions, used within a domain [Sie98]. Examples of these are Business Objects, Finance/Insurance and Manufacturing.

Current CORBA implementations include `Orbix` from IONA, `Visibroker` from Visigenic and `SOM` from IBM[10] [Szy98]. Very few implementations exist for the services and

[10]SOM follows the CORBA specification in some respects, but has added many of its own extensions so

facilities. It is in the nature of vendors to attempt to differentiate their product [Pri99], and most organisations will therefore use the same ORB throughout their organisation [AST99], causing vendor lock-in, which surely was not what OMG originally envisaged.

2.4.1.2 Middle-Tier Architecture

CORBA's (Versions 1 and 2) approach to providing the services required of the middle tier are currently based on the provision of an API. However, some CORBA vendors have incorporated runtime and deployment services into their CORBA implementations — even though this is not covered by the specification. These will often be offered to customers as an additional option, to enhance the scalability, reliability and availability of the product.

CORBA has recently released their *CORBA Component Model*, which changes their approach to providing services from the API to interception, the mechanism used by the other two component models [Ses00]. This interception is completely invisible to the component client, with all details being taken care of by the underlying architecture. They also provide a specification for an EJB/CORBA bridge, which allows a client to use a CORBA component as if it were an EJB component, and *vice versa*. At this early stage no implementations of this specification exist so it is difficult to tell how industry is going to react to this latest standard.

2.4.1.3 Example

CORBA version 2, for which current implementations exist, allows interfaces to inherit from multiple interfaces. Thus in CORBA we will define three interfaces for the example: the first interface `Customer1` for the first group of methods, the second interface `Customer2` for the second group, and the third interface `Customer` inherits from both of them. The client view is shown in Figure 2.13.

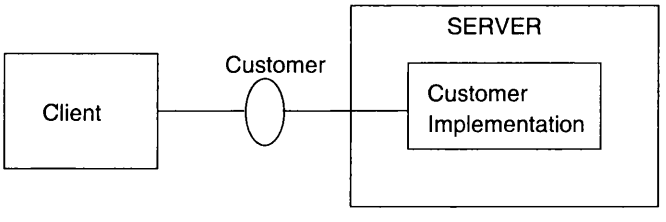

Figure 2.13: The CORBA Client View

CORBA Version 3, released in 1999, allows a CORBA object to have multiple interfaces. In this case, the `Customer` interface would not be necessary, and the clients would have access to either one, or both, interfaces for `CustomerComponent` .

that it is not a "pure" CORBA implementation [TMdlADF99]

2.4.2 Sun's Component Model

Enterprise Java Beans (EJB), the latest contender in the component-runtime middleware market, is Sun's specification for a server component marketplace [RS99]. EJBs were designed to support the development, deployment and management of transactional business systems using distributed objects built in the Java language [Kar99]. Sessions sees EJB as Sun's attempt to provide a portable virtual machine for the middle tier [Ses00]. This is because of Sun's focus on portability. The JVM, Sun's portable virtual machine, has been criticised on the basis of performance and functionality — at least on the user interface level [Ses00] — which may have contributed to the delayed uptake of EJBs in industry.

Programmers who enjoy using Java will like developing EJBs. However, this technology has been criticised for the same reasons that reservations about CORBA exist. People from several different computer organisations were involved in the development of the EJB specification, and it does seem to have the same sort of flavour of keeping everyone happy by incorporating all sorts of different features.

EJB is the youngest technology in the component-oriented middleware club, and it remains to be seen whether it will be able to perform as required in electronic commerce applications.

2.4.2.1 Architecture

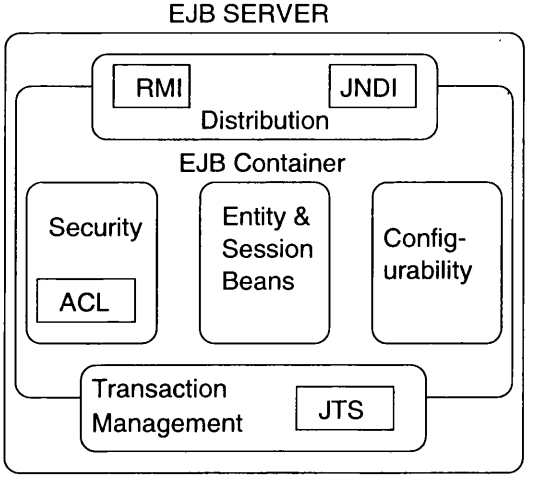

Figure 2.14: Enterprise Java Beans [PvV99]

The EJB component model [Tho97, Mic98a] is illustrated in Figure 2.14. All EJBs must be developed using Java. This language is used to develop both the interface, and the bean implementation. There is no interface definition language to be learnt by the programmer — as is required for CORBA and COM. The client locates the required beans by using the *Java Naming and Directory Interface* (JNDI) which provides a naming service. This is

layered on top of *Remote Method Invocation* (RMI), which is used to communicate with the server.

EJBs run within a component runtime environment called an EJB container. EJB containers are typically provided by any server container that satisfies the EJB specification. Transaction coordination is provided by the container interacting with the *Java Transaction Service* (JTS), which is essentially an implementation of the OMGs *Object Transaction Service* (OTS). All bean requirements, such as security, transactions and so on, are specified in a deployment document which defines the bean configuration requirements.

There are many implementations of the EJB standard. Examples are the Tengah server from Weblogic[11], Pramati from Proton[12], and PowerSystems from Persistence [13].

2.4.2.2 Middle-Tier Architecture

The container housing any EJB acts as an interface between the EJB and the client invoking the bean. Each bean will typically have two distinct interfaces, a `Home` interface (for managing bean instances) and a `Remote` interface (for application specific methods). This would seem to be a weakness of this model, since different roles could conceivably require more than two interfaces. There are two distinct types of bean:

1. *Entity Beans:* These beans are persistent objects, which model data in the underlying data tier. For example, a credit card bean would be an entity bean, because it is modelling the credit card data in the database. Each bean declares its requirements, for example, transaction isolation required, or security procedures, in a special descriptor object. EJB containers will provide the necessary middleware services automatically, as directed declaratively by the individual components. There are two kinds of entity beans:

   (a) *Bean-Managed Persistence* — persistence for this bean is managed by the bean itself. The bean must provide methods which will be invoked by the container to obtain data from the database, and to update this data when changed.

   (b) *Container-Managed Persistence* — The persistence for this bean is provided by the container. The deployment descriptor document will specify the linkage between the bean state and the underlying data structure; and the container ensures that the data is always consistent and correctly updated in the database.

2. *Session Beans:* A session bean models a business process, and executes on behalf of a single client. This could be a credit card verification, or a shopping basket for an Internet bookstore. There are two types of session beans:

   (a) *Stateful* — A stateful session bean will hold state between service requests.

[11] `weblogic.beasys.com`

[12] `www.pramati.com/products.htm`

[13] `www.persistence.com`

(b) *Stateless* — A stateless bean only offers a service. When a client uses a stateless bean the client has to take care of all state within their program, and pass references to the state to the bean to be operated on. This bean is identical in principle to COM+ objects.

2.4.2.3 Example

To implement the example using EJBs, the first thing to decide is the type of bean to be implemented. Since this bean will be modelling a database record, an entity bean will be used. The choice between bean-managed and container-managed persistence will depend on how complex the underlying data structure is. Since this data structure is very simple, container-managed persistence can be applied. This makes it far simpler to implement the bean, since no database access code must be written — everything is done by the container.

Every bean has two interfaces, so the `CustomerComponent` bean has home interface `CustomerHome` and remote interface `Customer`. The `CustomerHome` interface implements factory and finder methods, which will be used either to create or locate existing EJBs. The `Customer` interface will encompass methods from both groups mentioned in the introduction. The client view is shown in Figure 2.15. This means that any client programmer will be given access to the full functionality of the component, so that the security mechanism will have to be used to ensure that clients do not call methods they have no right to call.

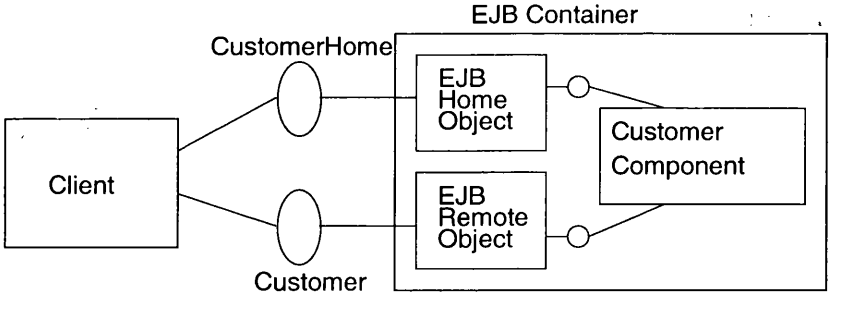

Figure 2.15: The EJB Client View

2.4.3 Microsoft's Component Model

This component model's main problem is its nomenclature. It started off as "ActiveX", which cannot be described as a descriptive name for a software component. Then the *Component Object Model* (COM) was introduced — and used two words in the name, component and object, which are meant to denote completely different concepts. The associated component runtime environment is called *Microsoft Transaction Server* (MTS), another misnomer, since it does not handle transactions at all. It is a component runtime environment, and delegates responsibility for transactions to the *Distributed Transaction Controller* (DTC).

Microsoft's component model was updated and the latest model, called COM+, was released in 1998. This is an umbrella name for many different products making up this component model. Having said this, it must be admitted that their component model is innovative and mature, and if Microsoft has its way, will dominate the middleware component market.

2.4.3.1 Architecture

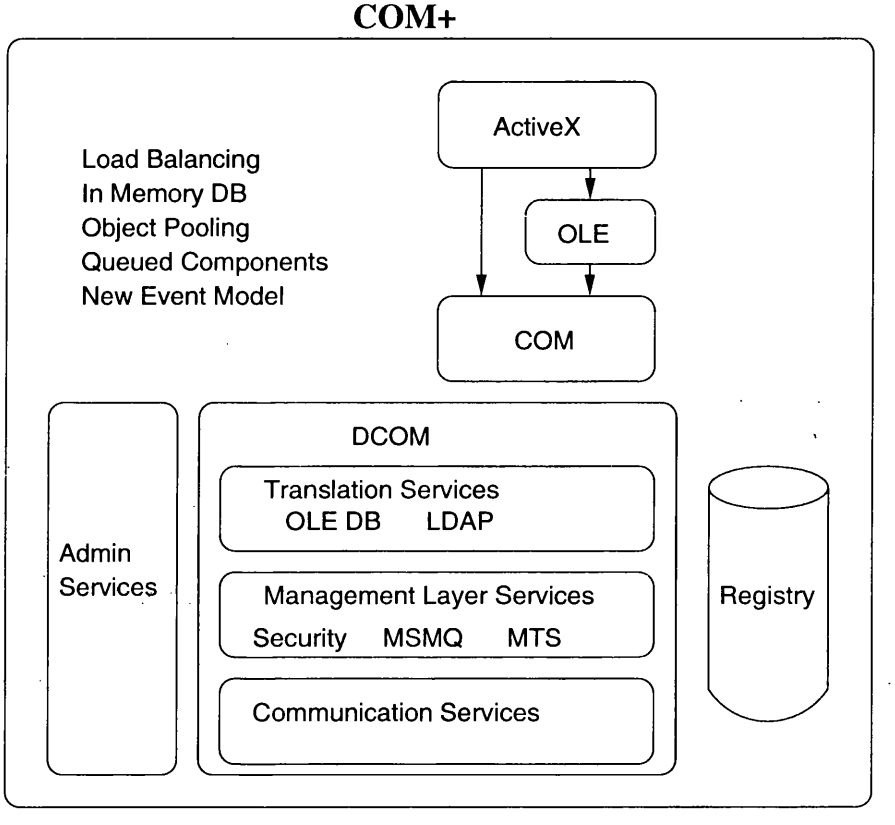

Figure 2.16: The COM Architecture [RE98]

To make things clearer, a list can be given, illustrated in Figure 2.16, of the main Microsoft Component Services [Raj99, TK98, RE98]:

- OLE — standard for compound document technology. The outer document acts as a container, while the other data inside the document act as a server. The server either embeds its data inside the document or links it — in which case the component will remain in its own file, with just a link being maintained in the compound document. Microsoft is not the sole vendor supporting this technology [Cha96]. The user of the compound document can edit the embedded component where it is — called either *in-place activation* or *visual editing.*

- OLE DB provides access to data in databases, files etc. It provides a set of classes

and interfaces which can be used by the developer to access the data which could be in various different formats.

- ActiveX — refers to the integration of components in applications. Examples of this are components used within web applications. ActiveX controls are generally used to display some visible entity at the user interface. They also have special methods which allow the client programmer to examine and set the values of certain properties which probably have bearing on their appearance.

  In addition, the ActiveX control also needs a mechanism which will allow it to communicate events to the client program. For this purpose the ActiveX control will have methods which allow its container (client program) to register an interest in certain events. When the event occurs, the ActiveX control will invoke a special method within its container to signal the event.

- COM — the integration infrastructure, used to implement components that interact either within a single address space, or between processes on the same host. It supports OLE and ActiveX, and other Microsoft Services such as DirectX. COM can be said to be the foundation on which all Microsoft's component software is based [Szy98]. It accesses other COM objects via interface pointers, which allows data and process encapsulation and transparent remoting. Its interfaces are immutable so an application with a pointer to an outdated interface will not fail because a new interface has been added.

- DCOM — extends COM to enable processes on different machines to interact.

- MTS — the component runtime environment in which components live, which watches requests coming into components and participates in processing them, providing security, automatic transaction management and a scaleable environment. The initial Microsoft component runtime environment, MTS, combines components with TPM capabilities.

- MS DTC — *Distributed Transaction Coordinator* which is very similar to CORBA's Object Transaction Service and which automatically handles distributed transactions.

- MSMQ — the asynchronous messaging capability which is somewhat similar to CORBA's dynamic invocation interface.

- LDAP — is an API which allows developers to access the registry. The registry stores information about the location of components, the users and groups in the system, passwords of those users, etc.

- Security Services — controls access to the system and the components a user can access.

- MS-RPC — is Microsoft's software which supports remote procedure calls. It supports DCOM's distributed processing functionality.

COM+ is a component software architecture that defines a *binary* standard for component interoperability. This means that the component will always fit into the required system correctly, since it is tailored to the underlying operating system. The components developed for the other two environments do not satisfy this requirement. It is difficult to take a CORBA component and plug it into another ORB, for example, and assume that everything will execute as before. EJBs too, are plagued by vendor-specific extensions, which means that an EJB developed using one type of EJB container will not necessarily fit into another container and work as before. At the very least the developer will have to import a different set of classes, and recompile in order to change containers.

2.4.3.2 Middle-Tier Architecture

The Microsoft component runtime environment is called *Microsoft Transaction Server* (MTS). It operates on an interception basis. Client requests will be intercepted so that the MTS can carry out various administrative functions. The functions offered by MTS are [RE98]:

- Administrative tasks such as monitoring transactions, performance etc.
- Resource management and pooling. This is essential for scalability and efficiency. For example, the pooling of database connections saves a great deal of time when accessing the database.
- Efficient triggering mechanisms — for example the "Just in Time" object activation.
- Support for asynchronous processing.
- Distributed Transaction Support. The MTS makes use of the *Distributed Transaction Coordinator* (DTC) to handle distributed transactions. This ensures that transactions which involve multiple data sources all commit, or all abort.

MTS currently only runs on Windows NT and 95. Services can be invoked from a browser, but only if the web server runs on a Windows NT machine. Microsoft achieves scalability by splitting data across machines and handling the distributed updates using MTS [RE98]. Unfortunately MTS does not support *Database Management Systems* (DBMSs) such as Ingres, Sybase or Informix, and does not communicate with DBMSs on other platforms.

2.4.3.3 Example

To implement the example, two interfaces could be defined for `CustomerComponent`, `ICustomer1` and `ICustomer2`, for the two groups of methods. COM objects can have multiple interfaces, so a new interface can simply be added when required. Administrators

could be given access to both interfaces, and give end-user client programs access to only the `ICustomer1` interface. This ensures that only the administrator can change client passwords. This means that each distinct feature of a component can have a separate interface. The client programmer's view of these interfaces is shown in 2.17.

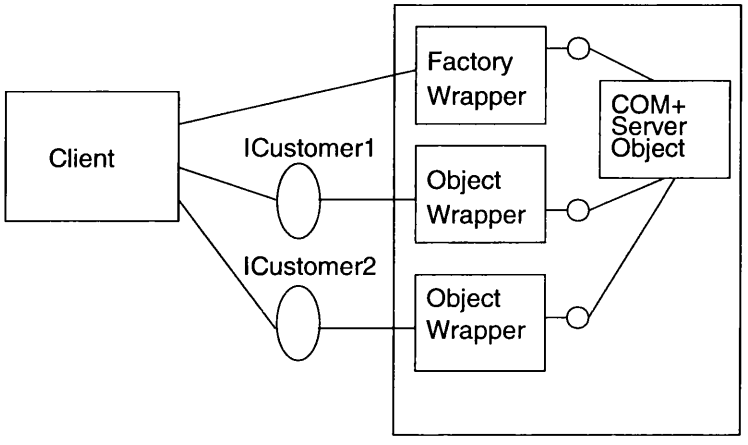

Figure 2.17: The COM Client View

2.4.4 Summary

Much has been said and written by experts claiming the superiority of one or another of these models. The table in Figure 2.18 summarises the basic differences and similarities between the three models. This section attempts to provide an unbiased comparison.

CORBA, the oldest component model, provides connectivity between application components, location transparency and many other middleware services such as naming, transactions, events, security and life-cycle management. Many vendors are only just delivering the implementations of the CORBA services. However, the CORBA specification does not address services such as load balancing, database connection pooling, result caching and failover. Consequently most of these features are not provided by most CORBA vendors. Some CORBA vendors do provide proprietary extensions to implement these services, but they are notoriously difficult to use [Dol98].

Many people criticise the Microsoft component model without really understanding it, and often only because they disapprove of the parent company's tactics. COM+ components are often criticised because they are stateless. Microsoft counters that this makes their middleware scaleable. COM+'s biggest disadvantage, the fact that it only runs on Windows 2000, has been turned to its advantage, because MTS is bundled with Microsoft's operating system. The other advantage is that it has been produced by the same company who made the operating system [Cha98]. COM+ is also said to be easy to use, and it allows application developers to use a number of languages, including Visual Basic, C++ and Java. On the

	COM+	EJB	CORBA
Interfaces	Many & Immutable	Home and Remote	Many
Transactions	Declarative	Declarative	API
Connection Pooling	Declarative	Declarative	API
Instance Management	Declarative	Declarative	API
Resource Sharing	Declarative	Declarative	API
Security	Declarative and Programmatic	Declarative	API
Link to Legacy Applications	CICS and IMS	Via JDBC	
Link to Other Component Models	CORBA	CORBA	Link to COM Proposed link to EJB
Component Platform	Windows 2000 IBM, some UNIX	Many	Any
Runtime Environment Platform	Windows 2000	Many	Any
Programming Language	Visual Basic C++, Java	Java	C, C++, Java Cobol, Ada
State Management	Stateless	Stateless or Stateful	Stateful
Naming Service (Component Instances)	No	Yes	Yes
Interface Definition Language	Microsoft IDL	Java Code	OMG IDL
Exception Specification in Interface Definition	No	Yes	Yes
Asynchronous Messaging	Yes	No	Yes

Figure 2.18: Differences and Similarities between the Component Models

other hand, Windows is sometimes perceived to be less reliable than Solaris or HP/UX, and is therefore less likely to be used for critical applications.

EJB, the newcomer, will be available on many platforms supporting Java, which means that EJB-based application servers can run on big powerful systems as well as cheap Windows systems. EJB also has some drawbacks. It fails to provide specifications for load balancing, directory services, distributed security services, and does not indicate which wire protocol should be used for controlling transactions. The EJB specification also allows vendors to add extensions to the API [Cha98]. This could invalidate Sun's claims of interoperability. Some people also feel that the restriction of only using Java could prove to be too much of a limitation. Supporters of COM technologies point out that it is unrealistic to expect one language to be all things to all people. They forget that COBOL did a pretty good job of this for many years, and is still to be found in many running systems today. Only time will tell whether Java will satisfy the needs of component developers to such an extent that other component models will overtake COM.

So, what is the conclusion? There is no winning component model. CORBA Version 3 provides a "Component Specification" which, among other things, provides support for a link to EJBs. It seems as though the OMG and Sun are joining forces to give Microsoft some much-needed competition. In choosing one of the three models, one has to take into account the platforms that the application servers will be running on, the criticality of the application, the programming language to be used, and the budget.

2.5 Component-Based Development

Component-Based Development (CBD) can be defined as:

> *the process of building systems by the combination, aggregation and integration of pre-engineered and pre-tested software objects* [Kar98]

thus providing a view of application development as an assembly process based on well-defined pieces of functionality [Bro99]. The original developers of component-based systems using generalised components simply glued chosen components together in a visual development environment. Unfortunately this only works for relatively small applications. Sophisticated applications need to have an application architecture, which has been arrived at in a new way, using a methodology matched to the special needs of component-based systems [Cha97]. Section 2.5.1 will discuss the special needs of CBD. Section 2.5.2 considers the possible sources of the component building blocks used in the CBD process, and Section 2.5.3 will enumerate the benefits that can be expected from this approach. Section 2.5.4 summarises this section.

2.5.1 A Different Approach

CBD requires a new approach. Whereas traditional monolithic software development followed the waterfall model, CBD needs an approach based on concurrency and evolution. Where traditional software development builds systems from scratch, or produces the system by modifying a previous system's code, CBD composes systems from pre-built components [Aoy98].

The creation of a software architecture for components will probably determine whether the system will be successful or a headache for the maintenance team. Bassett [Bas99] argues that there are two types of architectures in a CBD. The first set applies to runtime components, and the second to the parts used to construct those components. Execution architectures — for runtime components — can be divided into two layers:

1. *Technical architecture layer* — which technologies are used, how they fit together, and how they should be used.

2. *Application architecture layer* — how the applications look and feel to the users and how they should be broken up into modules.

Component architectures specify how the component can be customised and how it should be integrated into different contexts. Bassett contrasts execution and construction architectures by characterising an execution architecture as layering components to isolate independent sources of functionality or data, while construction architectures layer parts of a component to isolate independent sources of change.

Because components are essentially objects that have "grown up", object-oriented methodologies are easily extended to CBD. The waterfall model of software development has been rejected for CBS development, and the proposed methodologies suggest an approach based on iteration, incremental delivery and overlapping phases [Got98]. Tools for CBD need to support [BW98]:

- modelling of interfaces and component specifications;
- improved modelling for inter- and intracomponent dependencies;
- enabling component specifications to be developed independently of implementation details;
- new component-development approaches based on object-oriented analysis and design techniques.

Tools for CBD are beginning to appear. Some tools, such as Rational Corporation's tool which applies the *Rational Unified Process* method and ICON's *Catalysis* are simply extensions of their object-oriented tools. Other products such as Select's *Component Manager* and Sterling Software's *COOL:Spex* have been developed specially for the needs of CBD [BW98]. The latest tools support interface-based design as a key approach.

2.5.2 Component Sources

Component-based development rests on the notion of being able to procure the required components. It is necessary to distinguish between desktop components and middle-tier components, since the markets for these are very different. The current market generally caters for *desktop* components only — smart display-oriented components. Most of these are COM components — reflecting the overwhelming number of Windows-based computers on the desktop. Internet web-sites selling desktop components have sprung up over the last few years. Examples of these are Components Online[14] and ComponentSource[15].

There is, as yet, no equivalent market for business-logic type components. This could be due to the lack of standards for component description. Terzis and Nixon [TN99] propose a component trading facility which will support semantic trading within a community of component traders. They advocate the inclusion of non-functional information in component descriptions to engender and encourage component-oriented development.

Component buyers will have to ensure that support for the component will be available in the foreseeable future, so that they will be safer buying from established vendors rather than one-man businesses. Some software companies, such as IBM, Oracle, Amdahl, Fujitsu and Sterling Software offer specialised component groups to corporations, and some companies, such as banks, are considering selling their own specialised components [Mac99].

There are problems related to buying components, however. Components will have to be of high quality — or organisations will create their own and not bother to purchase them. The required quality can only be achieved if the customers are able to match their requirements to the stated capabilities of the components. Current practice merely lists interfaces with informal descriptions [Szy98], which is simply inadequate. Szyperski suggests that an explicit and unambiguous link is required between the component interface and its contractual specification to assist customers in choosing the correct components to meet their needs. There are alternatives to purchasing components, such as [WD98, Cha99c, SW98]:

- *Subscribe:* pay a subscription to make use of a remote component, rather than developing or purchasing a component for use in-house. An example of this could be a credit card validation facility.
- *Modify:* Develop a new component by altering an existing one.
- *Wrap:* Legacy components could be wrapped and used as components.
- *Develop* the component in-house and reuse it within the organisation. This is not as straightforward as it might seem, since a whole new programming paradigm must be introduced.

  For example, components are far more coarsely grained than traditional objects. Although the component's methods can be invoked by the client program as if they

[14] `www.components-online.com`
[15] `www.ComponentSource.com`

were locally available, the application developer *has* to remember that the middle-tier component could be located on another machine. Remote method invocations, while giving the illusion of being local, have a substantial time penalty attached. Szyperski points out that remote method invocations can be up to 10 000 times slower than local method invocations [Szy98]. Thus the object-oriented approach, which encourages the use of trivial methods like `getName()` and `setName()`, should not be supported by middle-tier components, since their use will increase network traffic unacceptably, and a significant performance penalty will be paid. To keep communication with the (probably distributed) component to a minimum, the methods should be such that all necessary information is conveyed together with a method invocation, and a significant service carried out by the component as a consequence.

A group of component vendors have recently formed a body — called the *Component Vendors Consortium* (CVC) which hopes to encourage the growth of a component market by developing standards of interoperability, documentation and technical support [Mac99]. This might be an important step in building a substantial component market.

2.5.3 Benefits of Using Components

The potential benefits of the component-based approach include [Kar98, Rog99, All99, Ses00]:

- *interoperability* — this is one of the main reasons that components made such a big impression in the first place. Components written in a variety of languages can work together to accomplish a common goal, often making use of diverse platforms. Before the advent of components, many organisations were reluctant to move over to object orientation because they would have to retrain all their staff in object-oriented techniques. Components can be developed in many languages, so the benefits of object orientation can be enjoyed without the rigours of retraining.

- *reusablity* — the same component can be used by many applications throughout the organisation, or sold to other organisations. Some organisations are already putting incentives in place to encourage reuse [Bae98]. There are also proposals to wrap legacy code and reuse it rather than re-develop. One of the greatest advantages of reusability is that code is well tested and problems are ironed out by long periods of use. Thus the product can be expected to be more reliable than code which is not intended for reuse.

- *control of complexity* — components separate the implementation from the interface, so that all actual implementation details are hidden. Components are also easy to understand, so that their use is not restricted to technical communities but is extended to business communities as well.

- *ease of change* — one component can be replaced by another, which implements the same interface, with the minimum of fuss. So long as the component adheres to the same "contract" published by the replaced component, the replacement will be unnoticed. In this category benefits such as maintainability, clarity and accuracy can also be included. This, in turn, leads to increased developer productivity due to a component's black-box design — the developer using the component does not need to understand how things are done.

- the *rapid development* of highly customised applications — components can be obtained from various sources to build an application, and customised to satisfy the application's particular requirements.

- *application reliability* — components should manage their own memory, resources and error management, but some may delegate some of this responsibility to the underlying operating system. Developers have to make provision for fewer of these issues, which should increase reliability of the entire system.

- *scalability* — the component runtime environment has been developed to take this responsibility, so that the component developer does not need to make provision for it — it happens automatically.

- *versioning* — some component models have built-in mechanisms which allow easy versioning of components. It is imperative that the holder of an interface to a component not be disrupted should the component be replaced, or upgraded. The old interface should still be supported, so that progress does not break existing applications.

These benefits, however, will not be automatically derived from making use of components. The list merely gives a flavour of the tremendous *potential* benefits of using components. Whether these benefits will be realised depends on many factors, such as the architecture of the application, the design of the component container architecture, and the quality of the available components.

2.5.4 Summary

It is worth reiterating what was said at the beginning of the chapter: software development is just as complex as it ever was. Some developers advocate the use of methodologies, while others feel that the "just build it" approach is better for projects with a short development time. Adding complicating factors such as distribution, parallelism and asynchronism to the software development process tends to make software development even more difficult. It is hoped that components will make this process simpler, but it does seem as if the "one best methodology" has yet to be found.

2.6 Review

Components, while solving many problems, have introduced a new realm of complexity into the lives of application developers. It is necessary to make an informed decision about components, and this section attempts to give arguments both for, and against, the use of components in systems development.

2.6.1 The good news about components

Software components have some advantages over object-orientation which will make software development simpler and go beyond object-orientation by doing the following:

- having interfaces which publish details about how to use the component, and specify which errors could result from the usage;
- reducing the scale of the unit to be produced by programmers;
- having standard ways of communicating with other components by means of method invocations. Method invocations were used previously, but their use is more ubiquitous since the advent of component technology.
- providing a better means for characterising components by their functionality in the application;
- providing a viable means for harnessing the functionality of legacy systems; and
- providing a better delivery mechanism than objects [SW98]. Objects (on their own), have never been reusable entities because they are often too tightly bound to other objects within a particular system. Current practice shows the reuse of *packages* of objects — the precursors of the current components. Components, however, *can* be reused because of their qualities of independent deployment and explicit context dependencies.

Many prominent people are firm in their belief that software components will be the way that software is going to be built in the future [ND99]. It occurs to us to wonder why it has taken thirty years for the revolution to happen. Reasons for this could be that:

- It has only just become clear to the software industry how the runtime infrastructure for these components should be built. The efforts of the members of the OMG, and the innovations of companies such as Microsoft, have made the acceptance and use of components possible, and financially accessible.
- The networks and communications industry has worked hard on solving the problems of communicating quickly and efficiently. This has made distributed applications the

order of the day — with distribution ceasing to be a complicating factor. Once distributed systems became common, it was only logical for organisations to want to use clusters of machines to load-balance, and they needed the capability to move software around easily.

- Three prominent component architectures have emerged and are competing for custom. This can only be beneficial since they will learn from one another and develop better products.
- The advent of the Web [WD98]. Components will be used to bridge thin Web clients to the traditional mainframes in many organisations.
- The growing legacy system problem. The fact that these systems can conceivably be wrapped and used as a system component is attractive.

Many organisations are throwing in their lot with component-based development [Bae98]. Large companies such as IBM and BEA are producing software to support the deployment of components, and companies such as Rational, Sterling Software and Sybase are offering component management tools to enable the development of component-based systems. This would seem to indicate that components are not simply a nine-day wonder, but something far more substantial.

2.6.2 Reasons for cautious acceptance of components

Even in the face of this progress, many organisations are not yet whole-heartedly embracing the new world of components. Kiely [Kie98] maintains that this is due to the fact that there are no standards for specifying component functionality and specific needs. There are also questions about how components should be billed for. In view of the fact that they are intended to be reused, component vendors might feel cheated at only receiving a single payment for a widely used component. Perhaps the widely used licencing systems would have to be engaged to bill clients on runtime usage of components. The cost of finding and understanding components, and tailoring them to specific needs, might prove to be cost ineffective.

Resistance to change could also be holding development teams back. Baer points out that developers who were disappointed by *CASE* are understandably reluctant to embrace this new panacea until it has proved itself [Bae98]. The other factor could be that management generally does not reward reuse, preferring to reward quantity of newly created code rather than reusable code [Gla98].

Chappell [Cha99c] argues that reusable business logic is just too difficult to create. This comes back to the point made in Section 2.1.1 about reusability minimising usage. The other difficulty with respect to reusability is that business logic changes so fast that the effort put into a truly reusable component might not pay off if the component is out of date in a

matter of months. Most experts agree that the one big factor standing in the way of wider acceptance of component-based development is a cultural one. It is undeniably difficult for programmers to put faith in other people's code, especially if this code happens to be perceived to be inadequately tested. Programmers routinely use other people's code when they make use of libraries, operating systems, and DBMSs, but these are all extensively used, and it can therefore be expected that any latent bugs will have been eliminated. If a programmer does not have this sort of reassurance about code, they are usually reluctant to trust it, and will rather rewrite it. The entire mind-set will have to be changed for component-based development to become the order of the day. However, the fact that the demand for new applications far exceeds the ability of programmers to supply this software may mean that programmers will simply have to make the cultural shift to components.

2.7 Conclusion

Organisations can hardly afford to ignore this latest innovation. Chappell [Cha99c], while expressing disappointment at the slow uptake of components, concludes that they are a crucial part of software's future. Many vendors have invested heavily in CORBA implementations, and many erstwhile TPM vendors have started marketing EJB Containers. However, most of these organisations have other products which could pull them through if component-based systems were to fail. COM+ however, is a critical and integral part of Microsoft's new Windows 2000 operating system. Microsoft therefore has a vested interest in making component-based development work [Ses99]. When Microsoft invests everything in a technology it is not going to go away. Component-based development is here to stay.

Having concluded this, it is necessary to acknowledge that component-based systems will, while solving a set of problems, create new anomalies. This dissertation considers one anomaly, the provision of adequate feedback to end-users. Those characteristics of components — their independent nature, third-party development and composition — which make them such an attractive option, are the very characteristics which make the provision of feedback to users more difficult.

Whereas feedback is a difficult problem to solve in any application, the distributed nature of component-based systems adds a new dimension to this difficulty, since it opens up a window of opportunity for a whole new range of possible errors. Application programmers need to account for these errors, so that when they occur they will be reported to the user in an understandable format.

In addition to this, it is necessary to consider the impact of everyday events such as interruptions on a user's application experience. If a system has not been designed with such events in mind, it will tend to disadvantage the end-user if use of the application is interrupted for an unspecified time period before resuming. All application user-interfaces need to be designed with the end-user in mind, and this includes planning possible responses to errors made by the user with great care. The following chapter takes a look at these

events, which here are called *quirks*, and analyses their effect on the end-user. Chapter 4 then addresses the general question of feedback, and considers the role of feedback in coping with quirks.

From then on, when anything went wrong with a computer,
we said it had bugs in it.
Rear Admiral Grace Murray Hopper, US Navy
on the removal of a bug two inches long from an experimental
computer at Harvard in 1945. (Time. 16 April 1984)

chapter 3

Quirks

The previous chapter introduced the concept of component-based systems, and concluded by arguing that:

1. the distributed nature of these systems made error reporting, with respect to system breakdowns, somewhat more difficult than for monolithic systems.

2. the possibility of interruptions should be taken into account when designing application front-ends.

3. the reaction of the application to user errors should be planned with forethought.

These issues are even more important in component-based systems, due firstly to the fact that the nature and experience of the end-user of these systems cannot be gauged as accurately as is possible in monolithic systems; and secondly due to the diffuse nature of these applications. This chapter thus introduces the concept of *quirks*, any occurrence which interferes with the normal execution of a task. A quirk is defined in the Oxford English Dictionary [SW89] as:

1. A sudden turn;
2. A trick or peculiarity in action or behaviour;

3. A sudden twist, turn or curve.

Section 3.1 will introduce the general notion of quirks, and Section 3.2 will provide a classification of quirks. Section 3.3 discusses the importance of quirks. Sections 3.4, 3.5 and 3.6 will describe the nature of each of the three types of events which cause quirks. Section 3.7 summarises the chapter.

3.1 Introduction

In executing a task, the user may take the direct route to proceed from beginning to end, as shown in Figure 3.1, moving directly from the initial state I to the final state F upon completion of the task. Using this direct path, with no detours on the way, is only *one* possible way of proceeding. In reality, this is a simplistic and unrealistic view of the way humans interact with computer applications.

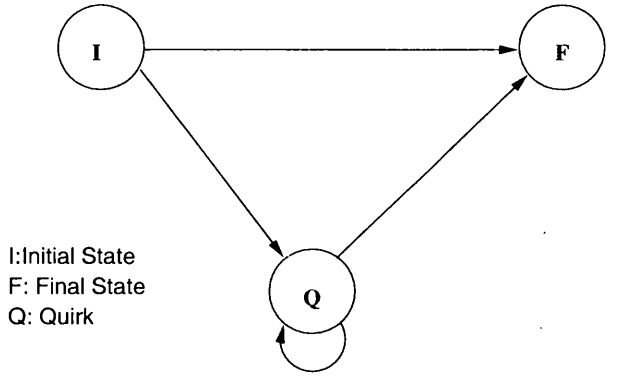

Figure 3.1: Initial and Final States in Task Execution

The execution of a task can be disrupted by a system breakdown, an error or an interruption — what will be referred to as a *quirk* — indicated by node Q in Figure 3.1. Simon [Sim69] points out that humans are basically serial in their operation, that they can process only a few symbols at a time, and that these symbols must be held in a limited capacity area (working memory) while they are being processed. Seen in this light it is not surprising that quirks can be so troublesome.

It is useful to build up a model of what quirks are and how users are affected by them. Simon [Sim69] states that a taxonomy can be seen as the first step in understanding a set of phenomena. Many researchers have worked on each of these different aspects — errors, interruptions and breakdowns — in isolation, but since there is often a commonality in the user's handling of each of these and in the effects on the user's emotions and task completion, it is useful to study them as forming part of group of similar concepts, as will be discussed in the following section.

3.2 Analysis of Quirks

The nature of these disruptive events will now be analysed to determine a commonality in the user's handling of the disruption. Jambon [Jam96] studied these issues and defined *singularities* to encompass the concept of a federation of the detection of human errors and interruptions, which can cause a user to suspend a task. Since the term "singularity" is somewhat ambiguous and since the intention here is to incorporate all errors (both detected and undetected) and also to include events such as system crashes, the term *quirk* will be used to refer to:

> *any event which causes the user to deviate in any way from the straightforward execution of a task.*

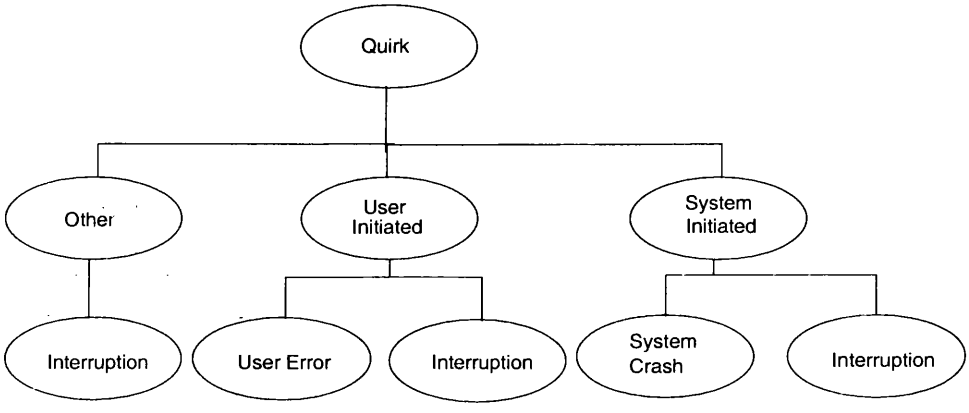

Figure 3.2: Classification of Quirks

Figure 3.2 gives a classification of quirks, which are a superset of Jambon's singularities. Quirks can be initiated either by the user, by the system, or by some external entity (*Other*). An external entity can interrupt the user's task processing by demanding attention elsewhere. The user could make an error, or interrupt the process voluntarily. The system could crash, or interrupt the process. Quirks are indicated by the node labeled Q in Figure 3.1. It is possible that more than one quirk will interfere with a user's execution of a task, hence the recursive arrow. The presence of a quirk could cause the system to end up in any of a number of different states, depending on the user's handling of the quirk. These different states will be explained in detail further on. The different types of quirks can be placed into one of three distinct categories:

1. *breakdown* — signaling a problem with some part of the distributed application;
2. *human error*;
3. *interruption* — this includes things such as external interruptions, user-initiated interruptions and system-initiated interruptions.

Before describing each of these quirks in detail, the following section will consider the question of why quirks are worthy of consideration.

3.3 Why Quirks are Important

Quirks are not merely an irritating fact of life, and should not be perceived to be purely a negative occurrence. Humans can only concentrate for limited periods before their brains become incapable of continuing without rest. Quirks can therefore be beneficial in increasing effectiveness and productivity by giving the user a rest. Research shows that certain types of quirks can raise worker stress and in some cases affect the health of workers. Quirks are worthy of some attention, because the extent to which the system designer develops the system with possible disruptions in mind will contribute to the usability of the system.

The critical point to consider is that a user who is busy with some activity builds up a *context* [Cyp86]. The context is a rich mental environment that stores all sorts of information built up during the time spent using that particular system to execute that particular task. Cypher points out that even a momentary distraction will cause this mental context to collapse. Czerwinski *et al.* [CCS91] have shown that advance warning of an interruption will enable the person to remember the context more effectively, and thus enable easier resumption of the interrupted task. People receiving unanticipated interruptions will tend to struggle more to re-establish their context upon resumption of their task.

Whereas quirks can have an effect on any user regardless of experience, the problem tends to be rather more serious for novice users. Novice users often experience a feeling of lack of control, fear and pressure when they have to use a computer or new application for the first time. Torkzadeh and Angulo [TA92] discuss the prevalence of computer anxiety amongst workers who first encounter computer technology. They point out that whilst computers have the *potential* for increasing productivity, reducing costs and gaining competitive advantage for an organisation, these advantages are not always actually realised for the employees. Users with the least computer experience have the most problems with computer anxiety. These feelings can only be exaggerated by error messages which the novice user often has no chance of interpreting correctly, let alone using to aid recovery.

Perry *et al.* [PSV94] found that a group of software developers spent 75 minutes per day, on average, in unplanned interpersonal interactions which makes these interruptions more common than might have been envisaged. Computer applications also build up contexts over time and could lose this context if suspended — unless the designer takes the possibility of a disruption into account when developing the system.

Brodbeck *et al.* [BZPF93] observed users handling errors when working with office computers. They observed negative emotional reactions such as anger, frustration and tension. Their findings are shown in Table 3.1. It is obvious from this table that reducing the time users need to spend handling an error can reduce negative feelings and lower stress levels.

Error Handling Time	Number of Errors	%Errors with Negative Reaction
Immediately	608	7.6
< 2 Minutes	330	15.5
< 5 Minutes	127	33.9
< 10 Minutes	11	36.4
> 10 Minutes	28	57.1

Table 3.1: Negative Emotions [BZPF93]

Fogg and Nass [FN97] argue that the rule of reciprocity, which exists in all cultures, also applies to human-computer interactions. As a consequence of this, users will tend to "help" computers that have previously helped them and retaliate against computers that have performed poorly. The frequent occurrence of errors would therefore tend to have far more long-term effects than merely the time spent in repairing the error would suggest.

Problems experienced with using computers have other negative effects on end-users. Studies by Yang and Carayon [YC93] have conclusively linked slow responses, breakdowns and insufficient information to increased worker stress. Schleifer and Amick [SA89] found that end-users became impatient and frustrated as a result of slow response times. End-users' health can also be affected, as shown in a study by Johansson and Aronsson [JA84] where a four hour breakdown was shown to cause an increase in blood pressure and adrenaline excretion. Lindstrom [Lin91] studied the effects of breakdowns and slow response times in office employees and found that they could be linked to excessive fatigue and nervousness. Waern [Wae89] cites research that has shown a connection between stress, dissatisfaction and frequent breakdowns.

There are also occasions when quirks have positive effects [Jam00, OF95]. One can hardly conceive of the fire alarm signaling a fire as a negative interruption[1]. Some system-initiated quirks are also helpful to the user. A virus warning is preferable to an undetected virus while a message informing the user of some event of interest can also be positive. We must therefore conclude that quirks are a fact of life and it is as well if they are accepted with equanimity and catered for by the application system.

It is necessary to understand quirks if one is to support the user when dealing with them. A categorisation of quirks has therefore been devised, in which they have been split into three broad categories — breakdowns, human error and interruptions. Breakdowns will be discussed in Section 3.4, human error will be discussed in Section 3.5 and interruptions will be described in Section 3.6.

[1] Fire alarm practice runs irritate and interrupt, but the benefit is so obvious that they, too, are positive quirks.

3.4 System Crashes and Breakdowns

The collapse of some part of the computing system will be referred to as a *breakdown.* Eldridge and Newman [EN96] studied the impact of technology failures on work. They identified so-called "agenda benders" — the effects of technology breakdowns which led to important activities not being completed on time. They found that the negative effect of an agenda bender, due to time lost in dealing with it, was exacerbated by the damage done to the rest of the day's activities. There was another knock-on effect, in which one person's technological problem had an effect on other people's agendas. They conclude that unreliable technology has a significant effect on work done during the day.

The rest of this section will address breakdown issues in three-tier systems. The type of problems which can be classified as breakdowns are a failure of (shown in Figure 3.3):

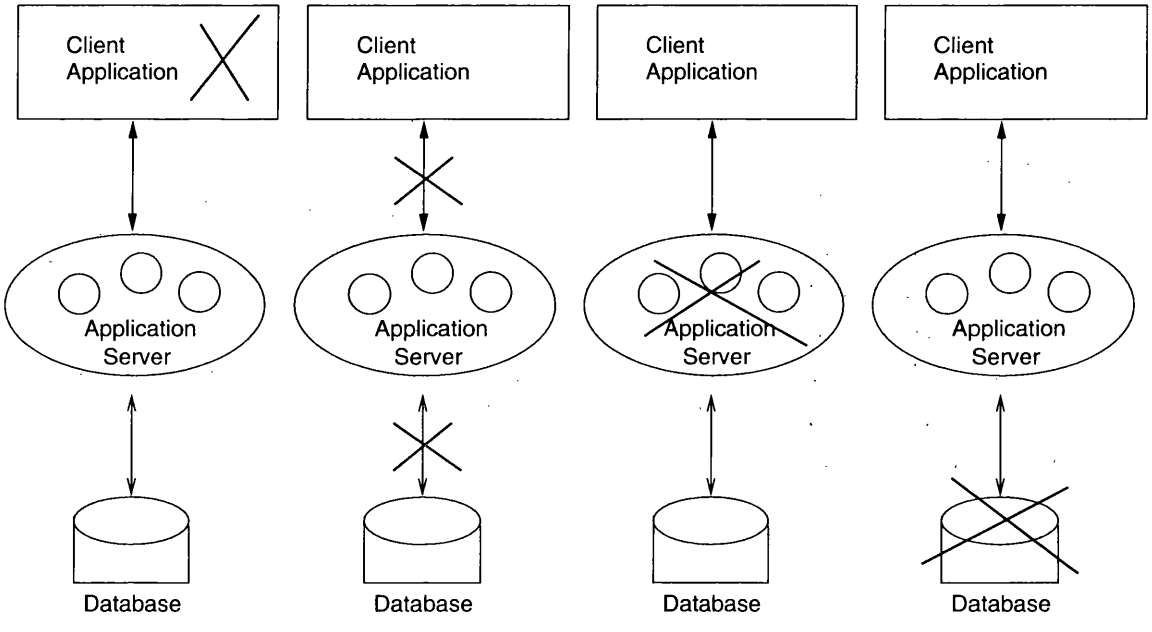

Figure 3.3: Breakdown Location

1. *the user's computer.* This would include moderate to critical failures — either of some application or of the whole computer.

2. *the network.* Networks can be affected by the following failures [Mul93]:

   (a) Crash — a faulty link stops transporting messages, but before stopping it behaved correctly.

   (b) Omission — a faulty link loses messages.

   (c) Arbitrary — a faulty link exhibits strange behaviour, perhaps generating too many messages or damaging messages.

(d) Timing — characterised by messages being sent either faster or slower than expected.

3. *the application server* —

   (a) failure of the server host, or

   (b) failure of the server housing the server component.

4. *the data store being used.* Since the application is completely separated from the data store by the middle tier, this type of failure will present as a failure of the previous type.

In the case of the end-user computer crashing, the user is generally left with little choice about how to handle the situation or doubt of its severity. After a crash, the user generally ends up in state IR shown in Figure 3.4 — the initial state reinstated after a recovery. This is not the same as the initial state I, since any application state built up before the crash will be lost and the user's context has been modified by the lost work.

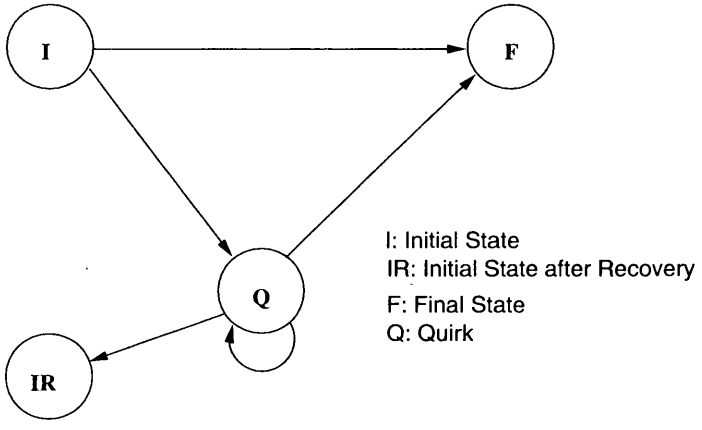

Figure 3.4: States in Task Execution, including state IR

In the case of a breakdown of the other computers involved in the distributed system or of the network, things become more difficult. The failure of some section of the system will mostly manifest itself by the reporting of an error by the end-user application. Sometimes the user will simply be faced with a lack of response from the computer, which could indicate a breakdown, but which could also conceivably simply be a symptom of an overloaded network. After a certain time period, the user will detect the problem and assume that the application has crashed. The rest of this section will therefore address the effects of breakdowns on the user — whatever their source.

The handling and effects of possible breakdowns can be classified on three axes — *extent, time taken to recover* and *assistance required* [Jam00]. The resulting graph is shown in Figure 3.5.

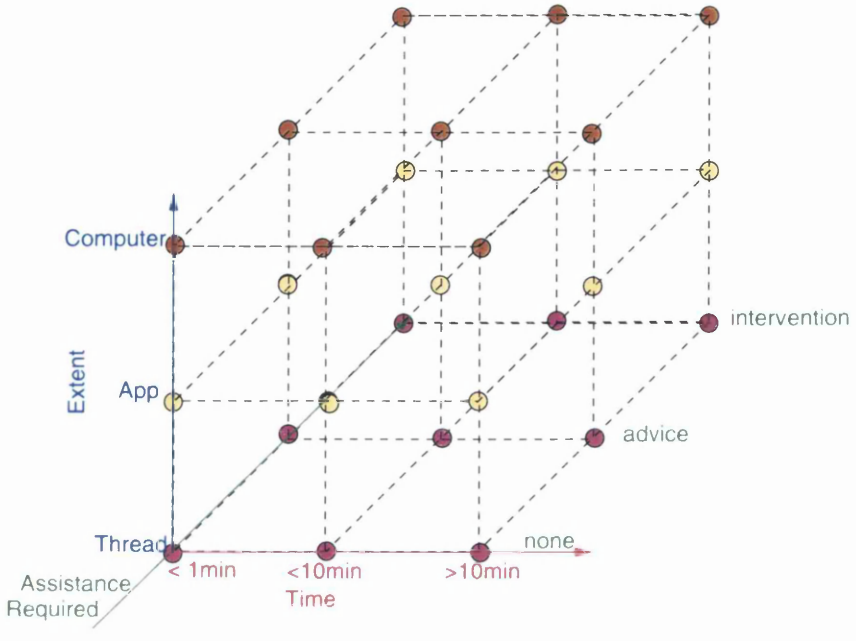

Figure 3.5: Classification of Breakdowns

Each of the axes will be explained in turn. The planes of the Y axis (labeled *Extent*), as shown in Figure 3.6, refer to the severity of the breakdown which is one of:

1. *moderate* — where the user's immediate process is disrupted. This is typically the failure of an application thread.

2. *severe* — where the user's entire task is disrupted. This is the failure of the application.

3. *chronic* — where the entire end-user computer crashes and no work can be done.

If the probability of each of these combinations is considered, the realistic planes become those shown in Figure 3.7. This is because a computer failure cannot realistically be resolved in less than 10 minutes and an application failure cannot be rectified in less than one minute. Intervention cannot realistically occur in less than 10 minutes, since presumably the user would have to summon assistance.

The X axis, labeled *Time*, refers to the time taken for the user to recover from the breakdown. This axis has three possible values, linked to the recovery from the disruption of the user's task. The values have been split up into the values of < 1 minute, < 10 minutes and > 10 minutes. This is due to the findings listed in Table 3.1, which show a sharp increase in negative emotions when longer than 10 minutes is spent in resolving an error. The different planes are shown in Figure 3.8. A more realistic view of the situation leads to the planes shown in Figure 3.9, due to the same arguments which limited the *Extent* planes.

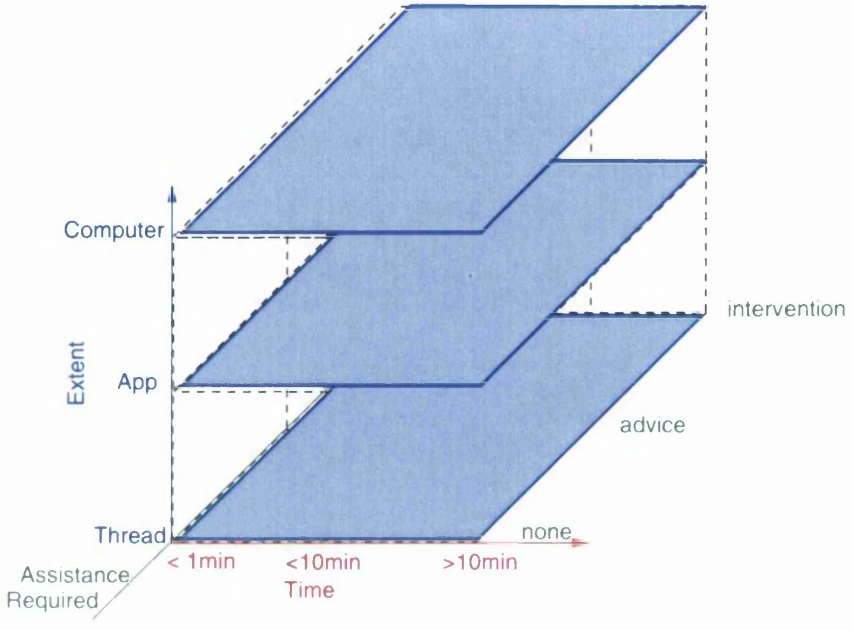

Figure 3.6: Extent Planes

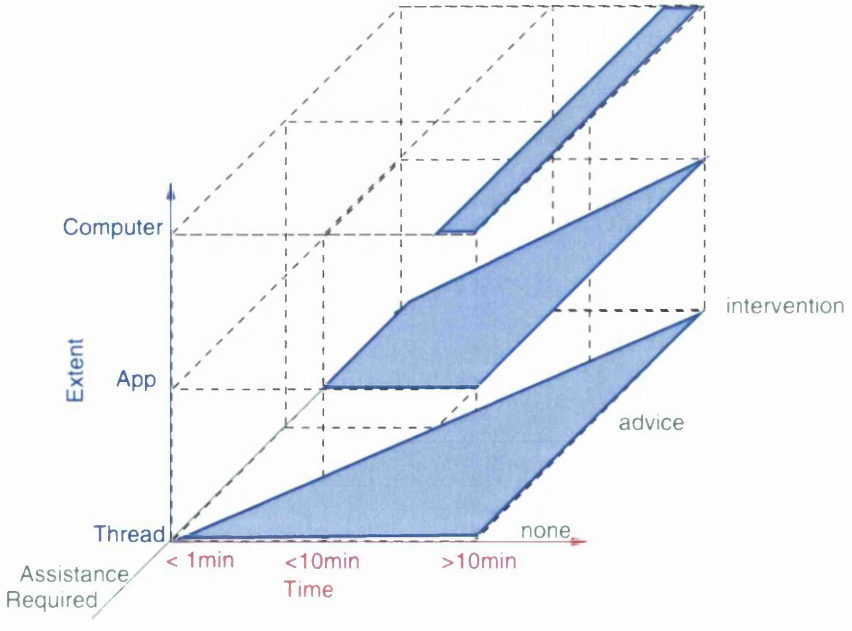

Figure 3.7: Realistic Extent Planes

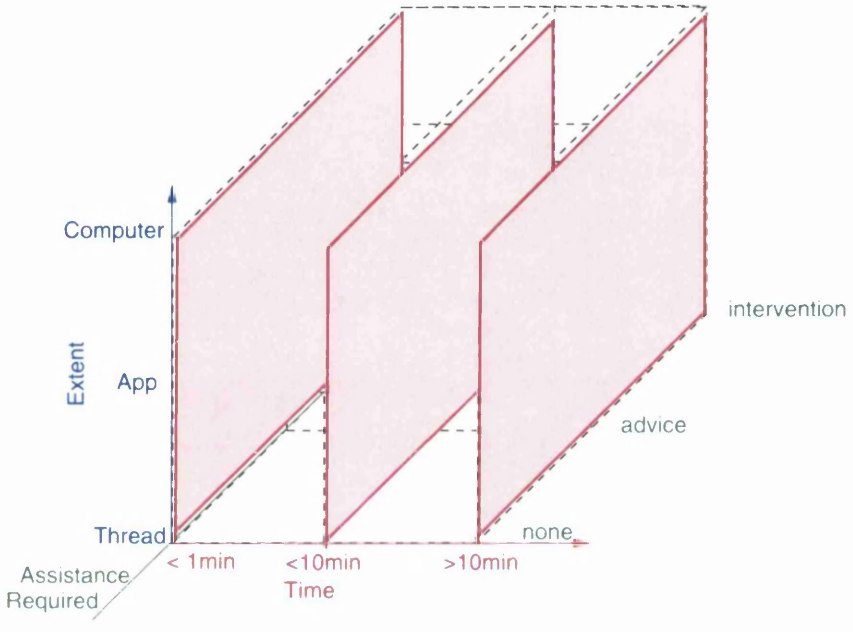

Figure 3.8: Time to Recover Planes

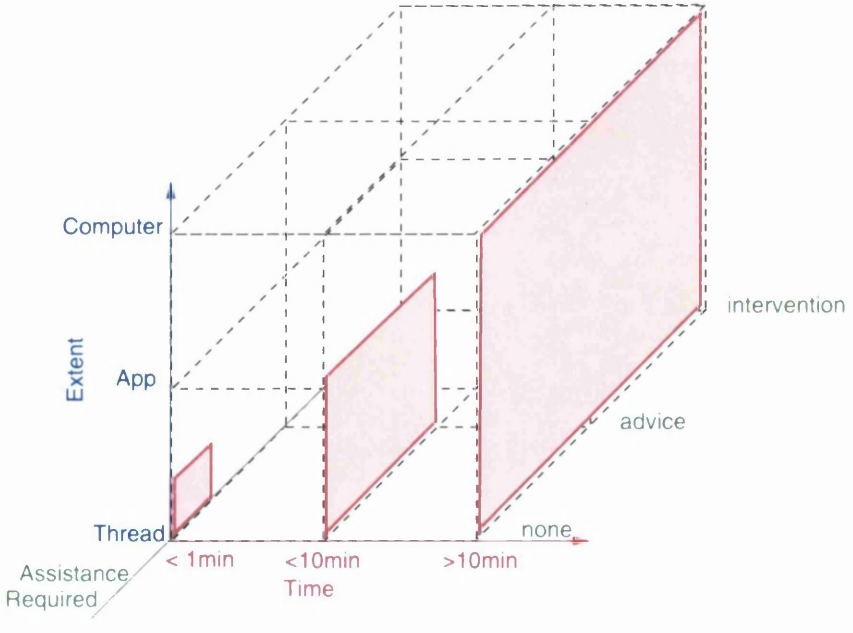

Figure 3.9: Realistic Time to Recover Planes

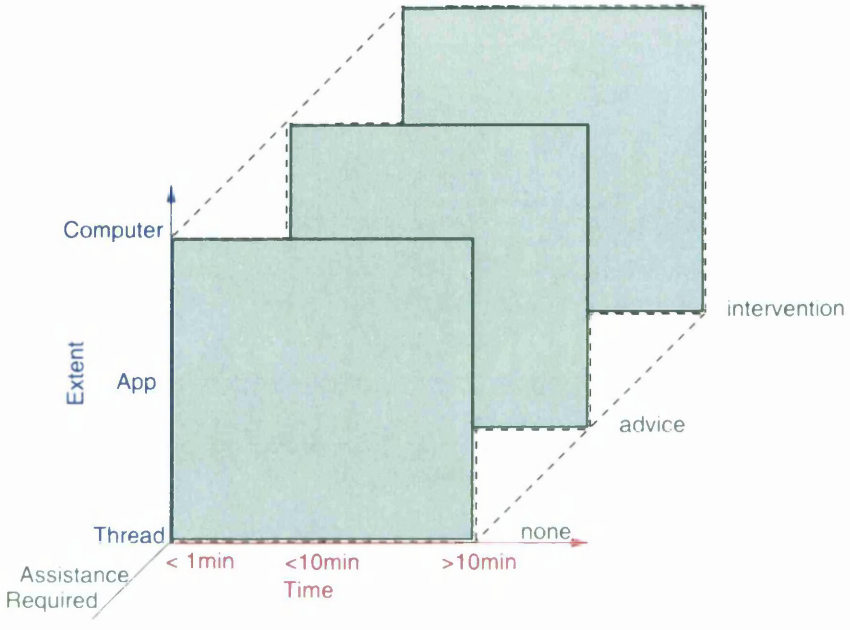

Figure 3.10: Assistance Planes

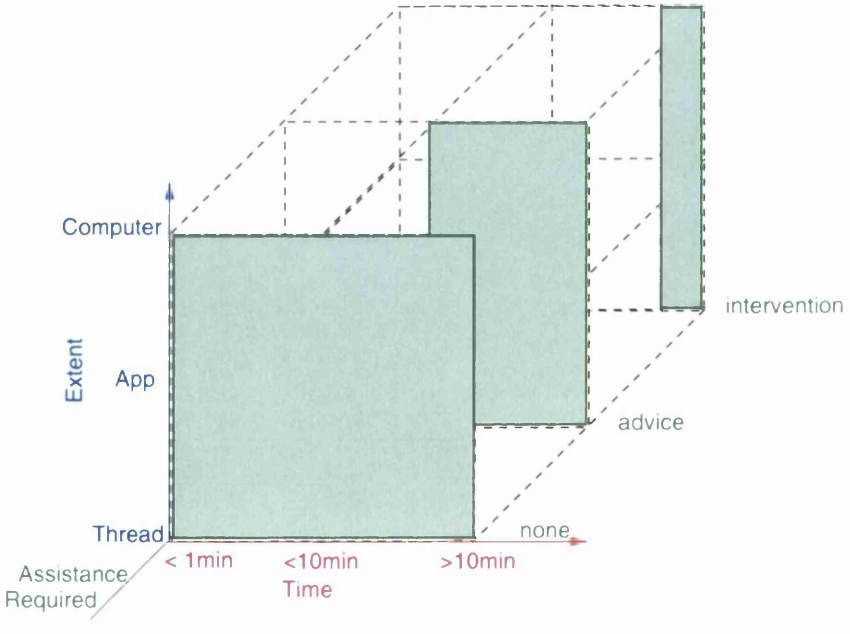

Figure 3.11: Realistic Assistance Planes

The Z axis, labeled *Assistance Required* — shown in Figure 3.10 — has three possible values:

1. The user will sometimes be able to handle the recovery from a breakdown — linked to value *none*.

2. The user may telephone someone for advice, or consult a manual — linked to the value *advice*.

3. When all else fails, the user may have to request *intervention* from a specialist.

Once again the planes can be limited as shown in Figure 3.11. It is simply not possible to get advice or assistance in less than a minute and intervention will probably take longer than 10 minutes to summon.

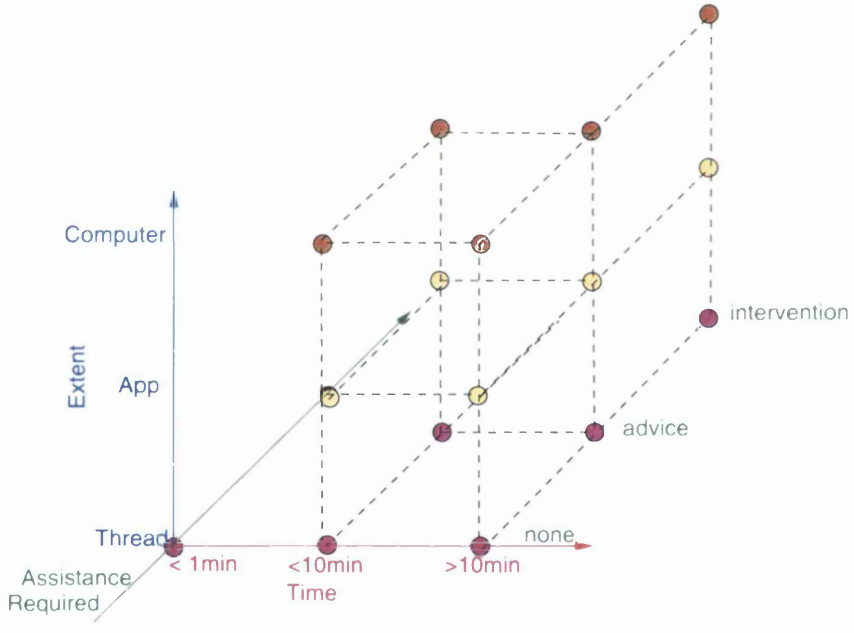

Figure 3.12: Classification of Probable Breakdowns

When all these restrictions are taken into account, the classification graph is reduced to the one shown in Figure 3.12. The obvious conclusion to be drawn from this graph is no surprise. The summoning of assistance from a specialist should be minimised so that the user's problem can be solved in the fastest possible time, thereby improving productivity and minimising stress. It is also obvious from the graph that breakdowns are almost certain to lead to negative emotions[2], something to which any computer user can attest.

[2]Since they will probably take longer than 10 minutes to resolve.

3.5 Human Error

Using a new computer application for the first time can be intimidating. This is especially true for non-technical people, but even applies to sophisticated users. The application developer faces an initial hurdle of getting people to use their application for long enough to overcome this initial period of uncertainty. Even after this period, it is possible for people to be put off by inadequate documentation or by the software overwhelming them with complexity [Bor91].

These problems are exacerbated when an error occurs. Errors are exasperating for novice users, but even expert users are not immune. Errors are always unexpected. The user is expecting to continue with the task, but now they are confronted with an error message which will require a completely different reaction. Surveys of computer use by expert users show that up to 10% of working time is spent handling errors [BZPF93]. Around 11% of successfully handled errors required external support. Errors are expensive in both human and economic terms. They contribute towards stress, interrupt the user's train of thought and can lead to negative emotions [ZBF+92]. The following sections will discuss issues pertaining to error handling.

3.5.1 The Nature of Error

The way error situations are handled is critical for usability. In the first place the user will probably need help in detecting and understanding the error; and, secondly, will probably not be able to continue using the system until the error situation has been resolved.

It is necessary to understand the nature of error, if there is to be any hope of providing help in dealing with the results of such errors. The next section discusses the types of error, while the following sections deal with the consequences of such errors. The following discussion draws heavily on the book on human error by James Reason [Rea90]. Reason considers the notion of errors in relation to intentions, since any attempt at defining human error must start with a consideration of the varieties of intention. Intention comprises two elements:

1. an expression of the end-state to be attained (the goal), and
2. an indication of the means by which it is to be achieved (the plan) in terms of one or more control statements (actions).

Once an intention has been formed and a *plan* formalised, the actions to achieve the intention are *stored* in memory and *executed.* Each of these cognitive stages (planning, storage and execution), has a related error type. Another way of looking at it is to identify errors which result from *intended* or *unintended* actions, the former being *mistakes*, the latter either *slips* or *lapses.* Mistakes are often referred to as *planning* errors, lapses as *storage* errors and slips as *execution failures.* These concepts are illustrated in Table 3.2.

Cognitive Stage	Error Type	Action Type
Intention		
Plan	Mistake	Intentional
Storage	Lapse	Unintentional
Execution	Slip	

Table 3.2: Error Types and Cognition

Slips are characterised by actions which differ from intentions. Slips are usually detected quickly since the state of the system is not what the user intended. The plan is usually correct, but the action fails to be executed correctly.

Lapses are due to a failure of working memory and short-term memory. Lapses include [Rea87a] forgetting list items, forgetting intentions, and losing track of previous intentions.

Mistakes are due to errors of judgement and reasoning errors [Rea87a]. Mistakes can be further classified according to the rationality that underlies them [Rea87b]. This classification relies on the notion that all human actions are governed by an interplay between the attentional and schematic modes of control:

- The *attentional* mode is a problem solving mode of control and is good at coping with novel situations but is limited, slow and laborious.
- The *schematic* mode of control makes use of inner "patterns" of action to handle situations for which a person has previously worked out a solution. The schematic database has no known limits and holds a vast number of "action patterns", each one of which fits a particular aspect of the world or skill the person has mastered.

Reason describes these schemata as large grain size action plans which are stored and which can be instantly retrieved for use. Kitajima and Polsen [KP95] contend that rather than a stored action sequence, the "stored" skills amassed by a person give the brain the ability to generate action sequences very quickly without conscious effort.

Whatever the mechanism, we can take it that there is a large body of knowledge at a person's disposal, which represents those tasks the person has mastered. This body of knowledge makes up a set of skills which can be used to carry out tasks and process information rapidly and in parallel without conscious thought.

As mentioned in the beginning of this section, the previous discussion of human error relied heavily on James Reason's analysis of human error as related to actions resulting from intentions. There is another perspective, not considered by his approach, which takes account of the fact that some actions may not be prompted by intentions but rather influenced by learned and subconscious behaviour. In stark contrast to the intention-based mode of operation is the undeniable fact that people often have subconscious reasons for their actions, and since the rationale behind their actions is often a mystery to the person

him or herself, let alone to others, they will often be at a loss as to the cause of the errors they will make as a result of learned or subconscious behaviour.

3.5.2 Performance Levels and Likelihood of Errors

Experts and novices make different types of errors because they are functioning at different cognitive levels. New users of a system typically have to invest a great deal of effort and thought into discovering how the system works. They have no internal "pattern" for achieving goals using the system. During this discovery period, they are essentially in a problem solving mode, which involves frequent decision-making episodes. When the user has learnt how to use the system and is a frequent and expert user of the system, many of the sets of actions required to achieve certain goals have become "automatic" and require little thought.

The artificial intelligence branch of computer science is based on the existence of *underlying plans* influencing user actions. An alternative view is that action is inherently *situated* — with plans having a limited prescriptive affect on user actions [Suc87]. The situated action view is that users react to their circumstances, with an objective in mind, rather than slavishly following some set of plans. This would appear to describe the nature of a novice's use of an application, whereas the expert's mode of working might be more aligned to Miller's plan-based mode of action [MGP60].

It should also be borne in mind that a user may be an expert at using some parts of a system and yet be a complete novice with features not used before. Thus it is not sensible to classify any user as wholly novice or wholly expert, but better to consider any user as ranging between these two extremes at any time during their use of the system.

During a study of 198 workers at 11 German companies conducted by Zapf *et al.* [ZBF+92] it was shown that experts committed many more habit errors than novices. Zapf's study proves that the nature of user errors changes and the help required by the expert user is consequently very different from the help required by the novice. Another study done by Kitajima and Polson has shown that slips are most often made by expert users and error rates for experienced users are found to be as high as 20% [KP95]. They are caused by the highly practiced, automated behaviour of the expert with the resulting lack of focused attention leading to a slip [LN86].

In the light of this, it would be beneficial to consider the error handling requirements of expert and novice users separately by seeing them as functioning at different performance levels [Ras87b, Ras87a]:

- The expert is engaged in a routine activity and the performance of an action requires successful retrieval from long-term memory. This retrieval is fallible and restricted by factors such as resource limitations and misperceptions. This results in errors, hence the high error rate for experts. The expert functions at Rasmussen's *skill-based*(SB) level. Activity at this level is controlled by know-how and stored automated schemata,

or rules. The conscious mind is often busy with other thoughts. *Slips and lapses* are generally made at this level.

- The novice user could be seen to be engaged in a problem-solving activity. The novice is trying to discover, by exploration, what the system does and this knowledge will be hard to acquire and difficult to attain [KP95]. The errors made by the novice user will therefore be due to a lack of knowledge about the underlying system. These users function at Rasmussen's *rule-based*(RB) or *knowledge-based*(KB) level. Rules or procedures are derived empirically during the use of a computer system. These rules are stored and while a user is using a system, information coming in will be seen as a signal, which serves to activate some predetermined rule. Mistakes made at this level, such as activating the incorrect rule, are called *rule-based mistakes.* If no rule fits, the user proceeds to the knowledge-based level. During unfamiliar situations, when no internal rule can be found to fit the situation, the person needs to develop a specific plan. Various plans are formulated and their effects tested against the goal, either conceptually or physically. Errors made at this level are called *knowledge-based mistakes.*

When a user is trained in a particular task, control moves from the knowledge-based or rule-based levels towards the skill-based level as the user becomes familiar with the system. The causes of mistakes made by the novice user are illustrated in Figure 3.13. The novice user is pulled by various forces:

- A: The rational thought processes, cognitively exhausting but capable of problem solving.
- R: The rules stored within the schematic database.
- E: Signals from the environment.

When the user is unable to deal with the forces coming from A, E and R concurrently, mistakes of bounded rationality occur. When the forces from A and R become confusing and the user veers between them, mistakes of reluctant rationality occur. When the wrong rule is retrieved from the schematic database, mistakes of imperfect rationality occur.

3.5.3 Detecting Errors

Detecting an error is the first step towards recovery. Mistakes are generally more subtle, complex and difficult to detect than slips. Slips are easier to detect because the action did not match the intention. Detection usually occurs as a result of comparing the outcome with the intention. Waern [Wae89] argues that user perception of performance is often defective due to inadequate feedback, or because the feedback is difficult to process.

After a *mistake*, the outcome matches the intention [LN86]. This means that it is hard to detect the error due to overconfidence, with the user using intelligence to explain

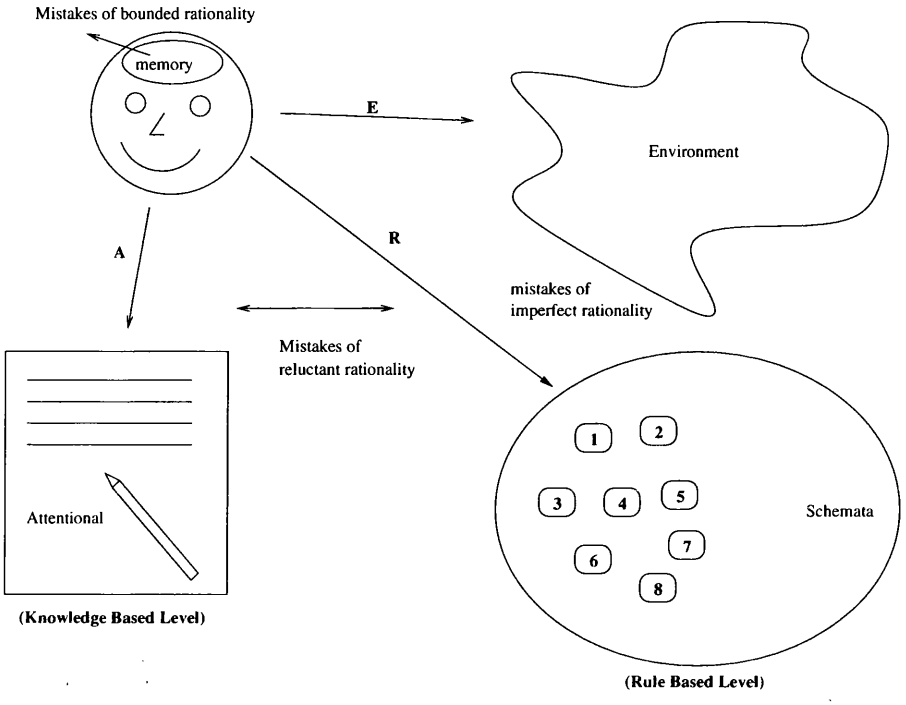

Figure 3.13: Shifts in Control causing Mistakes

away unusual occurrences thus failing to register the presence of an error. Overconfidence is caused by the person looking mainly for positive evidence of correctness. In research cited by Waern [Wae89], users were found to be better at detecting errors when they were directly instructed to look for negative evidence. Zakay [Zak92] has shown that immediate computerised feedback reduces this overconfidence level.

Errors are typically detected in three different ways: by self-monitoring; by some mismatch between what the user thinks the state of the system should be and what it seems to be; or by someone else pointing out the error. These all rely on some feedback mechanism — either by the computer or by some other means, allowing the user to compare what is expected with what has occurred.

A study of error detection during problem solving was carried out by Allwood [AM82]. Problem solving here is used as a blanket term including reasoning, judgement, diagnosis and decision making. Allwood instructed subjects to check completed work after finishing a task and found that the results of checking were either positive (satisfaction) or negative evaluation. Error detection occured during negative evaluation and involved two stages: triggering error detection, and taking steps to discover and correct the error. Negative evaluation was found to be of three types:

- *standard check* (SC): the subject simply decided to check their progress;
- *direct error-hypotheses formation* (DEH): triggered by a detection of a presumed error.

This implies actual detection of the error.

- *error suspicion* (ES): when the subject noticed something unusual and suspects an error. This suspicion does not imply actual detection of the error — merely a suspicion of error.

Once again the stored schemata come into play. Errors may be detected due to a mismatch between a stored representation and the currently observed error [Rea90]. On the other hand, the detection may be triggered by the subject's general expectations. The results of Allwood's study can be summarised as follows [Rea90]:

- Subjects had difficulty reacting to the effects of their errors.
- Among the types of evaluation mentioned above, DEH and ES occurred most frequently.
- Slips were detected far more readily than mistakes and most of the slips were detected by DEH episodes.
- The chance of successful error detection occurring during ES episodes decreased with the time elapsed between the error and the episode. This effect was noticed more with slips than with mistakes.

These findings suggest that improving the likelihood of error detection is by no means easy to achieve. Studies to measure error detection (full details in [Rea90], ch. 6) show that detection rates are 86.1% for skill-based errors, 73.2% for rule based errors and 70.5% for knowledge-based errors. The relative proportions of error types were 60.7% for skill-based errors, 27.1% for rule-based errors and 11.3% for knowledge based errors. This should not be misinterpreted to mean that KB errors occur least often, since it should be borne in mind that SB errors occur in the SB and RB levels, and that SB and RB errors occur at the KB level too [Rea90].

3.5.4 Enabling User Understanding of Error

Error reporting is far more effective if it is context sensitive. Hammond [Ham87] points out that interpretation of unfamiliar information makes heavy demands on working memory. An error message can be seen as an unfamiliar situation — at least to new users of a system. Thus it is to be expected that the user will be extremely likely to forget exactly what was being done prior to the error situation.

Most systems react to errors by generating error messages, but error messages are not necessarily the solution to the problem. The difficulty with error messages is well known, for instance [LN86, Nie93]:

- The format and tone of the error message is often offensive.

- The messages will often make people believe they have committed some serious error and that they are incompetent.
- Messages sometimes supply insufficient information and the user often does not know how to recover from the error.
- Messages often give obscure codes, instead of using understandable language.

It is important to remember that different users have different needs for error feedback — enabling understanding of error [MNG87]:

- Expert users often only need to be alerted to the fact that an error has occurred and to the location of the problem. Telling them the nature of the error is not important, since they can usually work this out for themselves.
- Frequent users, on the other hand, need to be told the nature of the error.
- Novice users need full explanations.

Understanding errors which result from learned or subconscious behaviour is far more difficult. The action which resulted in the error was *automatic* and the user may not have been fully aware of the action which caused the error. The user will probably need to be reminded of the preceding action, and then be given explanations in line with his or her experience.

3.5.5 Recovering from Error

Sometimes it is impossible to recover from an error. This is especially true of breakdowns. It is important that the user knows whether trying to recover is simply a waste of time.

The occurrence of a user error can cause the system to enter a number of states, as illustrated by Figure 3.14. There is a need to distinguish between system detection of an error and user detection of an error. This typifies the so-called "gulf of evaluation" [Nor86]. The width of this gulf is determined by the quality of the feedback in the user interface. (More about this in Chapter 4)

System Detection. If a user submits some input for a system to act upon, the system could detect an error and abort the action. The system needs to inform the user of the error with the success of the notification depending on the quality of the feedback and on whether the user is concentrating on the system at the time. If the user ignores or misses this notification and continues working, the gulf of evaluation has become wider and future actions will possibly be affected by this misunderstanding.

If the user does indeed realise that an error has occurred, either a decision can be made to abort the task — ending up at state IA (Initial State after an Abort) shown in Figure 3.15 — or to correct the input and continue working. Since the error was

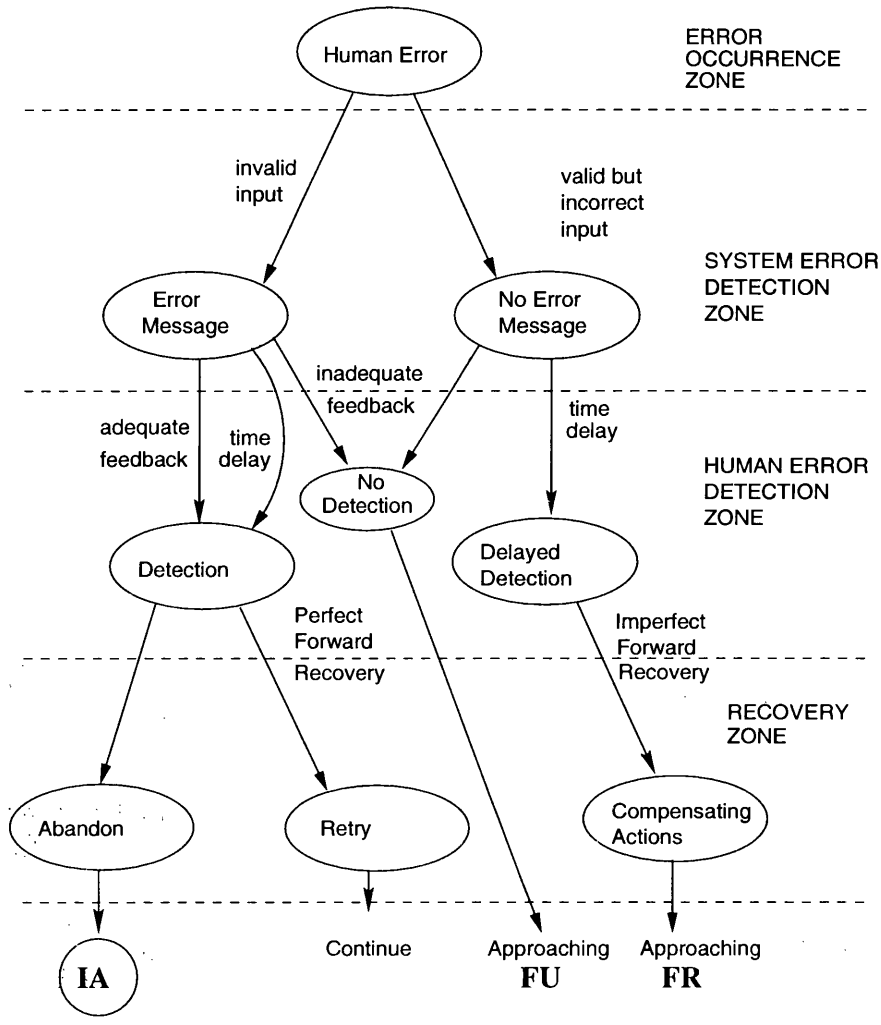

Figure 3.14: Analysis of an Error Occurrence

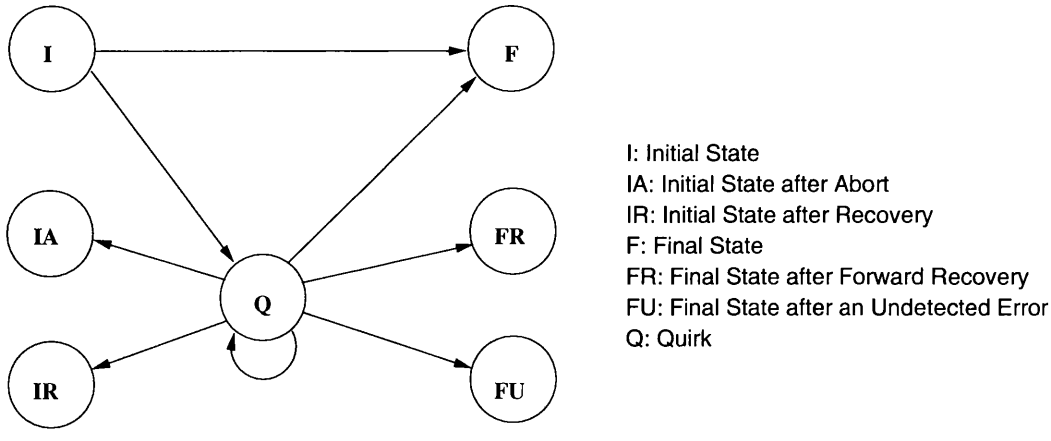

Figure 3.15: All Possible States in Task Execution

detected by the system, the effects of this error are not critical and the consistency of any underlying data store will not be compromised.

User Detection. If the user provides input to the system which is valid but not what they intended, the system has no way of realising that this is a mistake on the part of the user and accepts the input. The input will thus be processed and changes will possibly be made in one or more underlying data stores as a result. If the user were to discover the error, as a result of its effect, a decision could be made to supply inputs to the application which compensate for the error. The user could continue to work on the task in hand, but the final state will not be state F, but rather state FR, since another user could have made use of the incorrect information between the erroneous action and the compensation. If the user does not realise that an error has been made, then the gulf of evaluation, which has just become wider, needs to be bridged in order for the user to realise that an error has been made. The system is now in state FU, since the state of the system is not what the user intends and the consistency of the underlying data store has possibly been compromised.

Users sometimes do not realise that data was not useful till an indefinitely long time after the event. People also change their ideas about what was correct or incorrect over a period of time. People's memories are also notoriously inconsistent, even shortly after an event has taken place. The vastly differing eye-witness accounts of accidents are a well-known occurrence, indicating that people's perceptions of the same event are often coloured by inherent, subconscious factors beyond their control. In other words, the user may misremember inputs provided to an application, and accuse the "computer" of causing an error. The user may think that the inputs provided are correct, and only realise later, perhaps after speaking to a colleague, that he or she could possible have provided incorrect inputs to the application. It is often very difficult for users to check up on their actual inputs and interaction with an application once the application has terminated. It is also quite common for users to change their minds about the correctness of their actions over a period of time.

Application errors have purposely been omitted. These errors leave the application in an anomalous state and the user has no defence against them. They are not represented in the state diagram since they are almost impossible for the user to recover from.

The effects of user errors could accumulate, affecting the eventual recovery process and the error handling time, and exacerbating long-term effects of the error. The more unresolved errors in the system, the more time and effort will be taken to restore the system to the correct state.

Different types of errors occur for different reasons, because of failures at different cognitive levels. It is logical that the recovery needs are different too [BZPF93].

- In non-transactional systems, the undo function will work admirably for *slips* and *lapses*, for which the user is not exactly sure about what happened. As has been

explained, this is probably not an option in transaction systems. If the system detected the error, undo is not really necessary since the database has been unaffected by the error. If the system did not detect the error, undo is also not an option, unless the computer system is "intelligent" enough to generate a compensating transaction automatically. Thus, in a transactional system, errors such as slips, which are traditionally easy to recover from, become far more difficult to manage. This is because users realise that something went wrong, but have no idea what, since their actions did not match their intentions.

- Recovering from *mistakes* — rule-based and knowledge-based errors require complex actions compelling the user to go back through some actions to recover [BZPF93]. Users will often realise that something is amiss with their reasoning, or method of achieving the goal, but are at a loss as to how to go about recovering. Users, especially novices but occasionally also experts, will need external assistance to recover from such errors.

Knowledge activation and transformation are the crucial points which support the human error handling process [RPMB96]. Rizzo *et al.* argue that most mistakes depend on the mis-activation, conscious or unconscious, of knowledge. They further aver that error handling is the process of supporting the activation of relevant knowledge by modulating the conditions in which tasks are performed. It remains to be seen whether the mere re-activation of this knowledge and explanation of the effects of the error suffices to facilitate effective error recovery. Rizzo *et al.* propose the following guidelines for supporting the handling of human errors [RPMB96]:

1. *Make the action perceptible* — by this is meant that designers should make the match between action and outcome more obvious.

2. *Display the error message at a high level* — messages should be displayed at the user's level of understanding, with the possibility of getting more detailed messages should they be required.

3. *Provide an activity log* — thus supplying people with an external memory aid.

4. *Allow comparisons* — the user must be assisted in comparing the state with other, perhaps intended, states.

5. *Make the action result available to user evaluation* — this needs to be achieved as soon as possible. This aspect coincides with the discussion on feedback in the following chapter, which stresses that the feedback should provide aspects relevant to the task just performed.

6. *Provide result explanations* — the best way to provide error diagnosis is to give specific answers to the user. The user should not be overwhelmed by reams of explanations.

The user should only be given a high-level message, with further details available upon request.

3.5.6 Summary

This section has discussed human errors, their nature, their occurrence, their effects and issues with respect to user recovery from errors. Errors will be handled in the course of task execution and can be considered to be part and parcel of the task execution albeit an unpleasant or unexpected one. Error recovery can be likened to a "repair" effect often encountered in conversation. Listeners will give negative feedback if they either do not understand, or are not satisfied with what the speaker is saying. The speaker will then attempt a repair and get the conversation back on course.

3.6 Interruptions

Interruptions pervade our 21st Century lives. Telephones ring, people pop into the office and email continuously demands to be read and answered. Sometimes interruptions happen concurrently — for example, the telephone often rings just as you are about to answer the door. Often people feel that one interruption follows on from the previous one, leaving them no time to finish what they were doing. Humans routinely handle up to five activities simultaneously, and with ease, by interleaving them. Cypher [Cyp86] maintains that they do this by *linearising* — organising the parallel activities into a single linear stream of actions. Humans are very good at this — we have all seen evidence of this while watching someone cook a meal. The coordination of the various different activities, often while holding a conversation, is ample evidence of the versatility of the human race.

This interleaving of activities could be voluntary — such as when we decide that we do not want to wait for something to finish, and switch to another activity — or involuntary when, for example, the phone rings and has to be answered. In Section 3.3, the context which a user builds up during an activity was mentioned. In order for a computer system to support the user in linearising of multiple activities, it is essential that the user be provided with some sort of memory aid. This should keep the activity visible and provide a way for the user to "pick up the threads" as quickly as possible upon resuming an activity.

Care should be taken that the memory aid itself should not be distracting or clutter up the display. There is a continuous trade-off between providing the user with external memory aids and the limitations of working space [MN86].

Interruptions are common in the field of operating systems, with the definition of an interrupt being "events which modify the normal course of the execution of a program" [Kra88]. This definition could apply to errors and exceptions too, so it would be better to narrow down the definition a little. For the purpose of this discussion we will define an interrupt as being:

events, not caused by an error on the part of the user, which modify the normal course of execution of a task by a specific user using an application program.

3.6.1 Nature of Interruptions

A user in the process of using an application to carry out a task can be interrupted either by the application itself communicating some problem to be solved; or by something external to the application. This is illustrated in Figure 3.16. Each of these broad categories will be considered in turn.

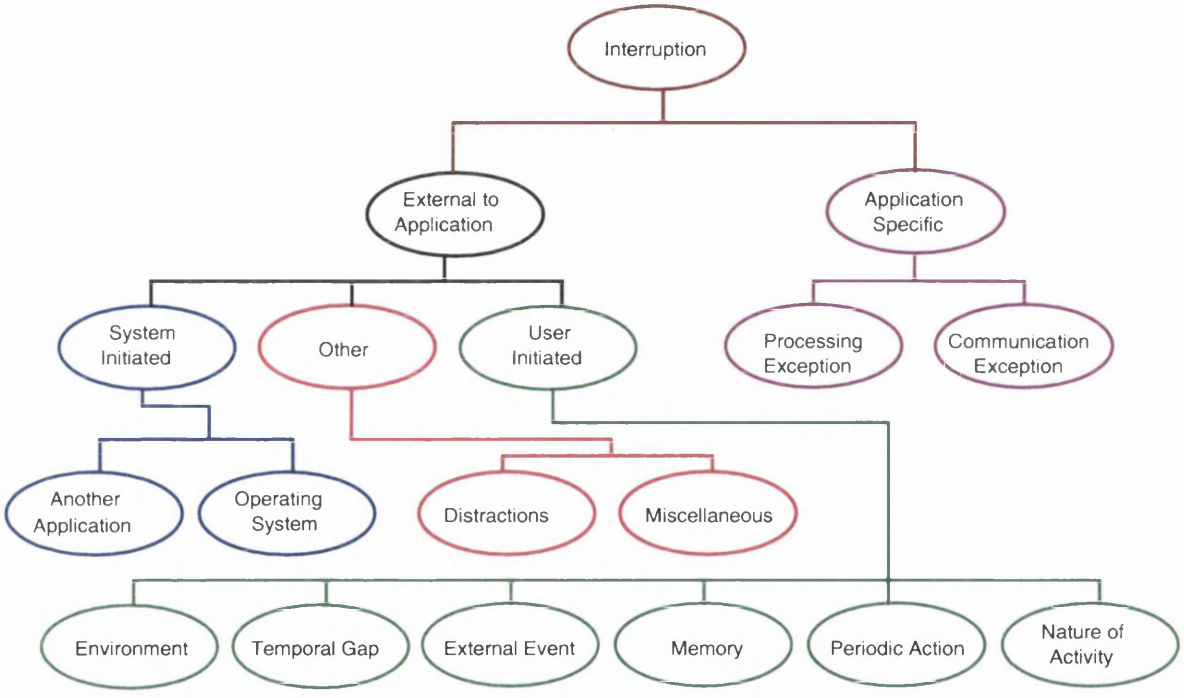

Figure 3.16: Classification of Interruptions

Application-Specific

This class of interruption will be split up into two distinct types: processing exceptions, and communication exceptions. These are different in that processing exceptions refer to exceptions generated by the application within itself, which have nothing to do with any communication with other parts of a distributed system. Communication exceptions, on the other hand, are received from some external entity. Since we are considering distributed component-based systems, this would indicate the failure of a global method invocation. The failure of a global method invocation will possibly indicate that the user needs to redo part or all of the actions which led to the method invocations. Note that breakdowns in the middleware or network are covered by the *breakdown* classification.

Application-External

This type of interruption could either come from the user's environment (external), or the user (internal). External interruptions come from our environment, while internal interruptions are caused by our own thought processes [MN86]. Application-external interruptions have been split into three types:

System Initiated — This type of external interrupt could come either from another application running on the user's computer, such as a mail reading program, or from the operating system itself indicating some sort of problem such as, for instance, a full hard drive.

User Initiated — This type of interruption is generated by user actions and could be triggered by one of the following external or internal factors [DRW95]:

- *Environment* — something external that reminds a user of something to be done. This could be a realisation that an error had been made of which the user has only now become aware. This could cause an immediate cessation of activity in the previous task in order to correct the error.
- *Temporal gap* — an expectation that something must occur within a certain time period.
- *External event* — for example, an alarm ringing to remind the user of an appointment.
- *Memory* — a memory of something that has to be done, or a need to check up on the activity of some other application. This is a prime example of an internal interruption.
- *Periodic action* — some actions are habitual and the importance of these actions could cause the user to interrupt the present task.
- *Nature of activity* — the interruption could be caused by the nature of the activity the user is engaged in, rather than some trigger causing an activity totally unrelated to the present activity. Cypher [Cyp86] cites the following mismatches between user activities and system programs, which lead to natural interleaving of actions due to internal interruptions:
  - *Single activity and multiple programs* — This would happen when some activity requires the use of more than one program. For example, someone sending an email might need to check a calendar to locate a free slot.
  - *Multiple activities and single program* — This occurs when a single program must be used for two different purposes in the execution of an activity. Programs such as browsers handle this type of thing quite nicely by allowing users to have more than one context (window) at a time, so that multiple

activities can be handled by the same program. Other applications are not as successful.

- *While-I'm-at-it activities* — These activities occur to the user in the course of some activity. For example, the user could be editing a document and, in the course of this activity, realises that there is no backup copy of the document on a removable disk. A decision could then be made to take the backup immediately, rather than risk losing the document.
- *Related activities* — A user sending an email message could need to incorporate part of a document in the message. This would require opening the word processing program in order to copy part of the document into the email message.
  * *Simultaneous interaction* — occurs when the user wants both activities to be visible at the same time, or wants to transfer data between them.
  * *Shared context* — is required when the user is perhaps using the same document in two different activities, for two different purposes. The system would ideally merge the editing from both contexts to arrive at the final document.

Other — This category includes two types of interruptions:

- *Distractions* — this is a special type of interruption. If the distraction is simply an irritating noise or a conversation between two other people in the same room, it requires no handling by the user, but does disrupt the task. The user has to acknowledge the existence of the distraction, change context to understand its content and then resume the original task, since the distraction does not require any processing. If enough distractions occur, the user could feel that nothing at all is being achieved. However, the user whose performance is degraded enough by distractions might feel the need to do something to handle it — promoting it to an interruption. The user may leave the room, or use ear plugs to screen out the noise, or even change task to one which does not require as much concentration. Gardiner [Gar87] points out that the immediate memory for visual abstract patterns, such as the structure and composition of a particular window, is disrupted by small amounts of distraction. With respect to verbal chunks of information (a familiar pattern — eg. a word or group of words combined according to a rule), short term forgetting increases with the level of distractions too. Research has shown that this finding can be applied to chunks of user actions within some action sequence, so that a distraction could make users forget where they were in an action sequence very easily [CCH00] — especially if no advance warning of the interruption was received.
- *Miscellaneous* — caused by personal visits or phone calls, or even the fire bell.

Some of these, such as a phone call, will allow the user to switch to the new context gracefully, with time to save context if desired. Others, such as the fire bell, generally do not allow graceful context switching.

3.6.2 The Composition of an Interrupt

The sequential structure of interrupts is shown in Figure 3.17. There are three sequential

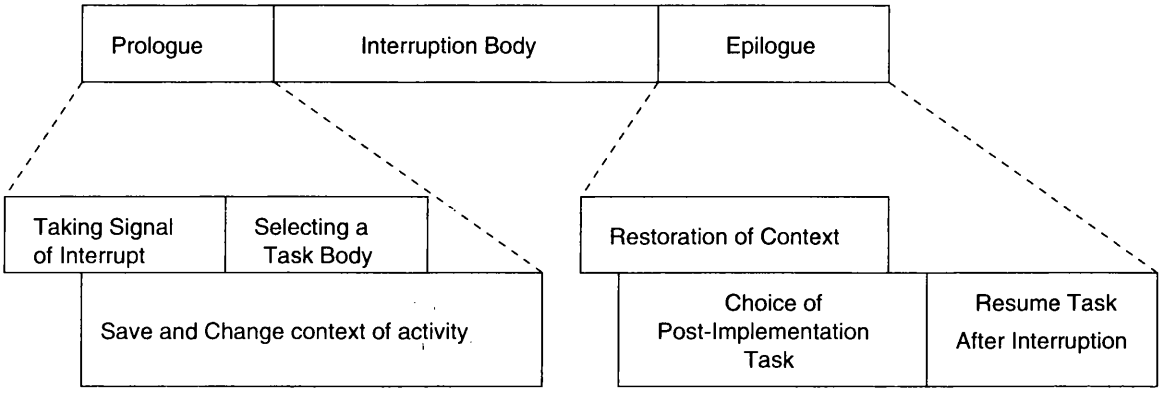

Figure 3.17: The Sequential Structure of an Interruption [Jam96]

stages, the prologue, the body of the interruption, and the epilogue. The three together make up the task interruption. Jambon notes that the body of the interruption is generally independent of the interrupted task, the "External to Application" in the classification. The classification also includes those interruptions which *are* dependent on the interrupted task. We consider that the application could publish an error message because of the failure of a method invocation which cannot be attributed to any error on the part of this user. The error could have been caused by the actions of some other user making use of the same middleware server, or data layer, and therefore cannot be classed as an error. This class of interruption is classified as an "Application Specific" interruption.

The prologue and epilogue are often dependent on the interrupted task. The user has to take some action in anticipation of handling the interruption. For example, the user may choose to save the document being worked on before answering the door. The epilogue will require the user to change context once again. The user has to try to remember what was being done and perhaps retrieve a document from the disk once again before resuming work. The epilogue could also lead to the user deciding to work on another task altogether — and not resuming the interrupted task. Waern [Wae89] notes that working memory is only able to retain information for a couple of seconds at a time and that unexpected interruptions can thus be fatal to an entire problem solving process.

3.6.3 Dealing with Interruptions

In the previous section the detection of errors by the system and by the user was analysed. This section will address the mechanics of handling interruptions.

Sometimes the handling of an interruption is interrupted by yet another interruption. Either the first interruption is suspended so that the most recent one can be dealt with, or the recent one is queued and forced to wait until the handling of the first one has been completed [WC95]. This mode of handling interruptions is defined by its sequential nature. However, the user may choose to interleave the handling of the interruptions, as is often done when a person suspends one phone call in order to answer another incoming call and then attempts to handle both in an interleaved fashion.

In some cases, the user will resume the original task, but in 45% of cases, according to a study done by O'Conaill and Frohlich [OF95], the user will not resume the disrupted task. This is illustrated in the diagram in Figure 3.18, by the transition to node O (Other activity state), instead of node F (Final state). O'Conaill and Frohlich tried to quantify the effects of interruptions in a working day. They found that the interruption was often seen to benefit both the initiator and the recipient, so that very few of those who participated tried to dissuade the initiator from making the interruption.

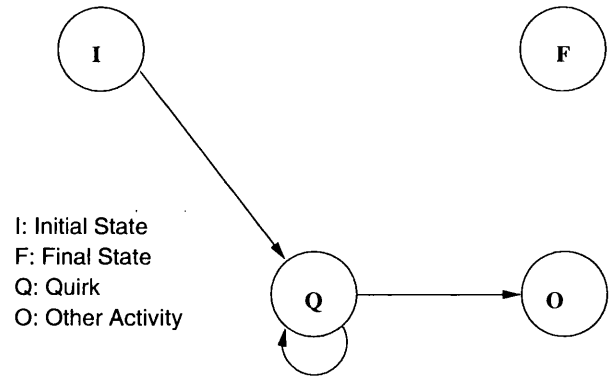

Figure 3.18: Non-Resumption of the Primary Task

Studies by van Solingen *et al.* [vSBvL98] into the effects of interrupts in software development found that the subjects of the study spent 1 to 1.5 hours per day on interrupts, and concluded that they spent up to 20% of their time servicing interrupts. The recovery time after an interrupt was gauged to be a minimum of 15 minutes.

Miyata and Norman [MN86] offer a perspective for understanding interruption handling by contrasting two types of processing styles that humans can be engaged in: *task-driven* and *interrupt-driven.* When someone is engrossed in some task-driven activity, such as reading a book, they will often screen out any interruptions they can in order to continue with the task in hand. When someone is doing a job such as answering a switchboard, they are in interrupt-driven mode. They are therefore tuned into taking interruptions and

dealing with them. The task-driven processing mode, when interrupted, will be difficult to resume because of the difficulty of resuming context, especially where the task involved a lot of thought. Interrupt-driven activity will, by its very nature, not be as negatively affected by interruptions.

Figure 3.19 depicts the interruption handling process (modified from [Jam96]). The way a user deals with interruptions is dependent on their present processing mode and on the perceived urgency of the interruption. A person who is in task-driven mode will probably filter out all but the most persistent of interruptions. Waern [Wae89] finds that people are able to eliminate irrelevant cues and thereby raise their level of performance. In this mode, they would probably choose to let email messages remain unread till they have completed their task. Someone coming into their office for help, on the other hand, will probably be "allowed" to interrupt their task. Thus, Jambon's [Jam96] model of interrupt handling in which the user either accepts or ignores the interruption, can be extended to take account of the two different processing modes.

As can be seen from the figure, the user can either filter out the interruption, or choose to take the interrupt signal. In the first case the user carries on with the task and the interruption does not disrupt that process at all. Miyata and Norman [MN86] suggest that people in task-driven mode are aware of the interruption, but not of the content. In the latter case, the user acknowledges the purpose and content of the interruption and chooses to either accept or deny it. If the interruption is accepted, the user needs to decide how the interruption will be dealt with and change context in order to deal with it. Hitch [Hit87] argues that the load on working memory increases directly in proportion to the amount of material that must be remembered temporarily or "held in mind". Consequently, the speed and accuracy with which people can process information will depend on the working memory load.

After the interruption has been dealt with, the user then needs to change context again and decide which task to proceed with. Often the nature of an interruption will determine the continuing activity. For example, if you are interrupted by a phone call telling you that your car has been stolen, it is likely that you would not continue with your previous task, but that you would phone the insurance company instead. Miyata and Norman [MN86] suggest that a system of reminders might be a good idea in ensuring that the user does indeed resume a suspended activity. Human memory limitations require these prompts, if a potentially critical activity is not to be forgotten. Simon [Sim69] (p40) cites two examples from literature about effects of interruption which show that while humans generally can retain up to seven units of information if there is no interruption between the encoding of the information and the recounting, they only retain two "chunks" of information after an interruption. Reason [Rea90] identifies errors of omission caused by interruptions, which in some cases have been the cause of major disasters.

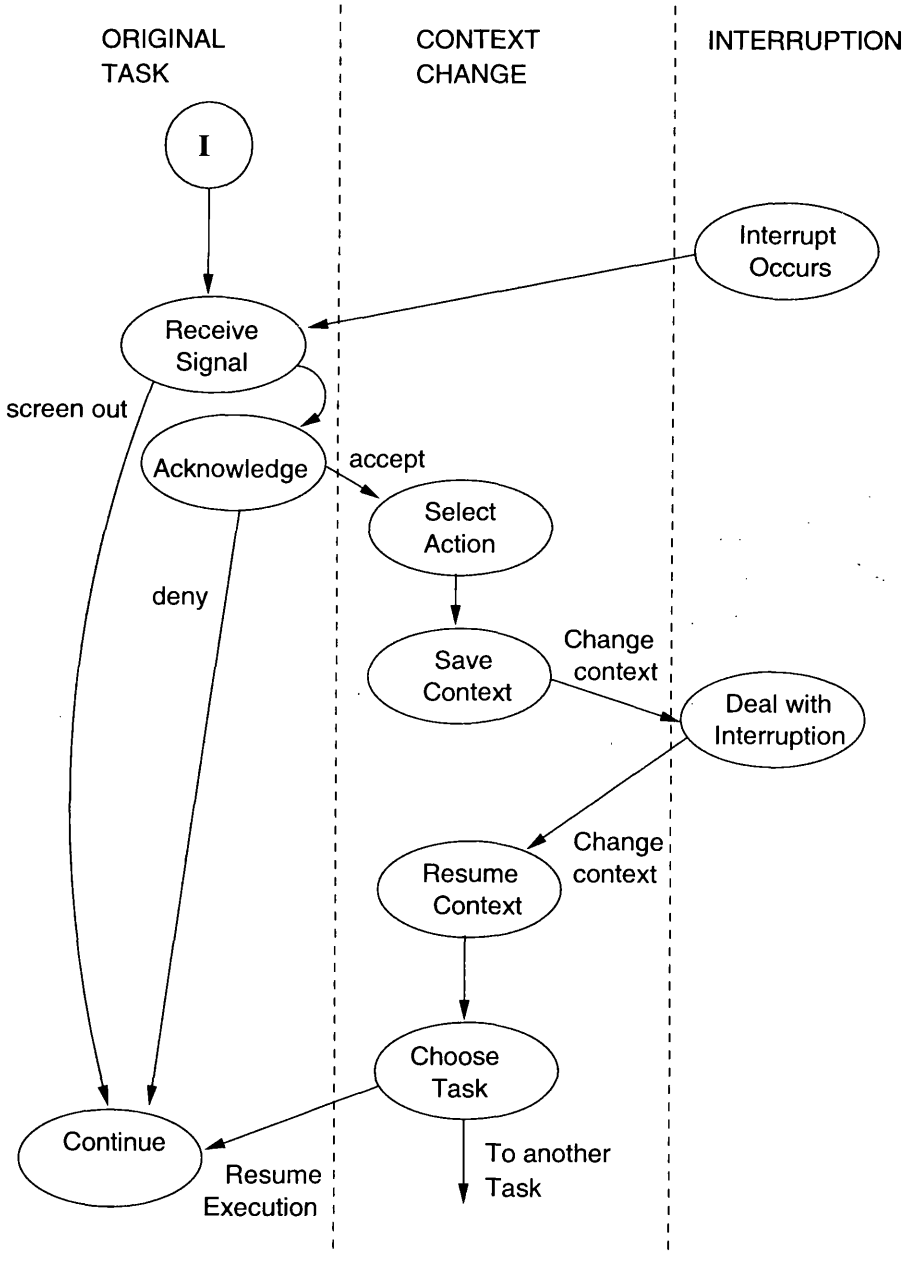

Figure 3.19: Interruption Handling

3.6.4 Summary

From the previous discussion it would appear that the biggest problems confronting users who are interrupted are firstly that they might forget what they were doing; and secondly that even if they do return to the original activity, they have difficulty rebuilding the context, or train of thought. In order to assist the user in recovering from interruptions, it would thus be helpful to have the following features provided by the application:

- mental aids, to help the user remember past actions;
- graphical features to allow the user to take a couple of steps back to rebuild the mental context.
- user assistance in building an awareness of the history of interaction with the application, by linking past inputs to the results — or outputs – thereof.

Since each user has different "remembering" needs, the principle of giving the user an overview and then allowing zooming-in to get required detailed information, applies here.

3.7 Summary

This chapter has investigated quirks in some detail. Their nature has been explored, as have their negative and positive effects and techniques for dealing with them. This dissertation intends exploring the role of an enriched model of feedback to, amongst other things, assist in alleviating the negative effects of quirks. Such feedback would have to provide information about current application activity as well as a sense of the user's past interaction with the application. The following chapter will introduce the feedback concept in general terms, and then concludes by discussing how feedback can be used to alleviate the negative aspects of quirks.

Ah! What is man? Wherefore does he why? Whence did he whence? Whither is he withering?
Dan Leno (George Galvin)
Dan Leno Hys Booke. (1901) ch.1

chapter 4

Feedback

The work presented in this dissertation attempts to provide a general purpose feedback framework for applications that use the component technology surveyed in Chapter 2. Further chapters will discuss this problem with specific reference to component-based systems. This chapter concentrates on the role of feedback in more general terms and refers to the role that feedback plays in alleviating the negative effects of quirks.

The need for feedback, the means for providing it and the difficulties inherent in this, from a programmer's point of view, are discussed in this chapter. Section 4.1 explores the nature of feedback and likens it to human conversation. Section 4.2 examines the use to which people put feedback. Sections 4.3 and 4.4 discuss the motivation for, and timeliness of, feedback. Section 4.5 examines aspects of good and bad feedback and makes recommendations about the type of feedback that should be provided. Section 4.6 explores the concept of the provision of feedback graphically rather than textually and Section 4.7 considers the role that feedback can play in alleviating the negative effects of quirks. Section 4.8 summarises the chapter.

4.1 Introduction

Feedback is a word widely used with different meanings in several academic areas including engineering, economics, biology, mathematical models or biological systems, formal logic

and social science [Ric91]. The Oxford English Dictionary [SW89] defines feedback as:

1. The modification, adjustment or control of a process or system (as a social situation or a biological mechanism) by a result or effect of a process esp. by a difference between the desired and an actual result.
2. Information about the result of a process, experiment, etc.
3. A response.

Spink and Saracevic [SS98] argue that all academic perspectives have a basic concept of feedback as *involving a closed loop of causal influences, a loop of mutual or circular causality.*

The research in this dissertation focuses on the human-computer interaction perspective, which has an interest in *"the exchange of information between participating agents through sets of channels, where each has the purpose of using the exchange to change the state itself or one or more others"* [Sto94]. The feedback thus concentrates on the method and type of interaction, the participants in the interaction, their purpose, and the interface between the human and the computer [SS98].

Shneiderman [Shn86] defines feedback as *communication with a user resulting directly from the user's action.* Pérez-Quiñones and Sibert [PQS96] point out that this definition does not cover communication from the system which notifies the user about the state of the system, or feedback about long-lived activities or transactions. Feedback allows the computer to fulfill the same role as a conversational participant. Suchman [Suc87] points out that the immediacy of the reactions of computers of today, combined with the fact that such reactions are not random but considered (having been designed by a human programmer), lead people to consider the computer to be a purposeful social object — a conversational participant. Friedman and Millett [FM95] found that people, even computer literate people, attributed social attributes to computer technology.

Participants in a conversation do not merely take turns, but in many ways collaborate in the conversation. The person doing the talking expects a level of feedback from the person being addressed, either in the form of nods, verbal affirmations ("uh huh"), or facial expressions. These indicators are used so that the person doing the talking can determine whether the person being spoken to is receiving the message and understanding it. A facial expression can convey a negative response to what a person is saying, or a positive response, indicating understanding. A blank expression, on the other hand, could indicate that the person being addressed is deaf, or does not understand the language of discourse. Thus the feedback can be seen to be either positive (when things are going smoothly) or negative (when the listener signals a problem), and the feedback will determine the future conversation [BH93].

This conversational feedback model neatly fits in with the interaction between humans and computers. Clark and Schaefer [CS87] proposed a model of collaborative contributions to conversation and identified four possible states of the person being addressed:

1. not aware of being addressed;
2. aware, but did not hear what was said;
3. heard it, but did not understand;
4. heard, and understood;

If we apply these principles to the human-computer conversation, it is logical to assume that the user will want to be able to identify these states in a conversational partner — the application — so as to know how to proceed. Nickerson [Nic76] tries to pin down the nature of the participants in human-computer interaction. He points out that in normal conversations one would not call one or the other participant a user. He argues that in human-computer interaction this nomenclature is correct, since the human-computer conversational model, although in many aspects similar to the human-human conversational model, is quite different, since the human partner can be characterised by their goals and cognitive abilities, while the computer cannot.

Suchman [Suc87] likens the human-computer interaction to human-to-human conversation in three ways. Both are:

1. *reactive* — Computers react to user actions, meaning that the control of the human-interaction process is essentially in the hands of the user.
2. *linguistic* — The use of computers today is not a matter of pulling levers and pressing buttons, but rather of specifying operations and considering their results — exhibiting linguistic behaviour.
3. *opaque* — The general opacity of human participants in a conversation makes explanations about human intentions critical in understanding human action.

Suchman argues that the aforementioned reactive, linguistic and opaque properties of computers lead users to attribute *intentions* to the behaviour of the computer system. She further argues that, having drawn this conclusion, users then expect the computer to explain itself and expect it to behave in a rational way.

When humans communicate they often assume a shared background knowledge of the particular subject they are discussing. The speaker will have to gauge the listener's understanding of the topic and take steps to explain further if the speaker concludes that the listener does not have some knowledge needed in order to understand the conversation properly. The listener will also make assumptions about the speaker's knowledge and opinions during the conversation. Suchman points out that much of what is said often requires reference to other facts which are unspoken, but relevant to the conversation. As a conversation continues, the two participants will learn much about each other, their knowledge, attitudes and expectations.

The success of human-computer "conversation" will depend on the user being able to gauge the "knowledge" of the application — and being able to supply the computer with those items it needs in order to continue the conversation successfully. Feedback is a valuable tool in the hands of an application developer, who needs to communicate the application's "knowledge" and expectations to the user to facilitate the application's role as conversational participant. In conclusion, perhaps the best definition of feedback would be

> *the communication of the state of the system, either as a response to user actions, to inform the user about the conversation state of the system as a conversation participant, or as a result of some noteworthy event of which the user needs to be apprised.*

4.2 Purpose of Feedback

The previous sections have justified the need for feedback and discussed issues pertaining to the timeliness of feedback provision. Before proceeding to further examination of feedback provision and attempting to compile a list of desirable feedback features, we need to take a look at the *use* to which the user will put the feedback which is provided.

Feedback was defined at the beginning of the chapter as achieving the following (somewhat paraphrased):

1. *signifying a response.* This serves to reassure the user and confirm that inputs have been accepted and that the system is acting upon them.

2. *modifying the behaviour of the user.* If we once again consider the similarity of the human computer interaction to a conversation, feedback will serve to help the user decide how to proceed. Without feedback, either negative or positive, the user is left wondering whether to pursue the original course of action, or to veer to one side or another to accommodate some fault.

   Engel and Haakma [EH93] distinguish between two kinds of feedback which are pertinent here — *I-feedback* and *E-feedback.* I-feedback refers to the reception of information already supplied by the user while E-feedback communicates to the user the next inputs expected by the system. Whereas I-feedback is genuine feedback, E-feedback is considered to be feed-forward, *affecting future behaviour.* Engel and Haakma argue the importance of E-feedback, since it reveals the system's expectations and allows the user to judge whether these expectations are compatible with envisaged intentions.

3. *promoting understanding.* The user needs to understand the system and the effect that inputs are having on the state of the system. Without a good understanding both of the present state and the role the user has played in bringing the application into that state, he or she cannot hope to proceed knowledgeably.

In addition, feedback can be used for

- *overview purposes.* The feedback could be used by some other application, as is the case with application monitoring, or by some distributed entity which needs to monitor performance of the application, or by the user to provide some information not pertaining directly to the state of the system, but rather to other characteristics such as performance, workload etc.

The traditional role of feedback is often seen as pertaining exclusively to the first use mentioned above. There is a need to widen that view to encompass the other uses, in order to provide a complete feedback mechanism. These are not traditional uses of feedback and the following sections will address the extension of the feedback concept to include these features.

4.3 Why give Feedback?

De Bono [dB98] points out that it is often better to simplify a process than to train people to cope with complexity. Feedback can be considered as a way of simplifying the interaction between the user and the system. To justify the "simplifying" role of feedback, it is necessary to understand the nature of the interaction between the user and the computer.

One of the first attempts to model human interactive behaviour was done by Card, Moran and Newell [CMN83], who proposed the GOMS (Goals, Operators, Methods and Selection) model. GOMS is a very good model for predicting temporal properties, but not as good at accommodating the effects of human thought [Dix91]. Norman's action-based approach [Nor86], which analyses the interaction between humans and computers, identifies the stages of human activity shown on the left hand side of the following table, while the matching stages of conversational activity (since we are considering feedback needs with respect to conversational dialogue) are shown on the right:

Step	Stages of Human Activity	Conversational Stages
1	Establishing the goal	Establishing the goal
2	Forming the intention	Deciding what to say
3	Specifying the action sequence	Formulating the words in the mind
4	Executing the action	Saying the words
5	Perceiving the system state	Hearing the reply
6	Interpreting the state	Understanding the reply
7	Evaluating the system state	Interpreting what was said

During this activity, Norman identifies two gulfs that have to be bridged as a result of the difference between human goals (in psychological terms) and system states (expressed in physical terms). The two gulfs which need to be bridged to enable human use of a system

are the gulfs of *execution* and *evaluation.* The gulf of execution represents the effort that the user has to make in order to translate goals into action sequences which, when applied to the system, will achieve the goal. The gulf of evaluation represents the effort the user has to make to understand the state of the system as a result of their actions. Norman argues that these gulfs can be bridged from either direction. The system can narrow the gulf by constructing the interface with the needs of the user in mind. Norman notes that the user can bridge the gulf by creating plans, action sequences and interpretations of the system.

There are two different schools of thought with respect to the motivation behind user actions. The artificial intelligence branch of computer science is based on the concept of the existence of *underlying plans* influencing user actions. An alternative view is that action is inherently *situated* — with plans having a limited prescriptive effect on user actions [Suc87]. The situated action view is that users react to their circumstances, with an objective in mind, rather than slavishly following some set of plans.

Clancey [Cla97] explains that the theory of situated cognition claims that what people perceive, how they conceive of their activity and what they physically do, develop together. He adds that human action is essentially improvisatory by directly connecting perception, memory and action, concluding that conceptual knowledge is developed over time as part of, and by means of, physical performances.

The conversational model of user interaction in the current paradigm of recognition rather than recall [Dix91], seems to lean towards the situated action perspective, rather than a plan-based mode of operation as proposed by Miller *et al.* [MGP60]. Users behaving in this manner are even more dependent on the narrowing of the gulf of evaluation, since they react according to the way they interpret the state of the system. Dascal [Das92] argues that the structure of dialogue is inherently reactive, with the speaker planning what to say in reaction to what was said (according to the current state of the dialogue).

The traditional plan-based approach suggests a fore-knowledge of the application's user interface. The expert user may indeed have this knowledge, but the novice or occasional user would tend to react to the state of the application rather than act according to some set of plans. O'Hara [O'H94] suggests that neither plan-based nor situated action would suffice to describe all interaction. He suggests a continuum between the two modes along which people shift according to factors such as knowledge and task.

The quality of the feedback provided by the system can go a long way towards narrowing the gulf of evaluation — in conversational terms, enabling an understanding of what was said. Feedback becomes very important when the system is prone to long response times, which often happens in distributed systems. A slow response could be indicative of an error or simply a normal occurrence if the network is overloaded. Either way, the user needs to be fully informed about the reason for the delay. Feedback becomes critical in the case of system failure. Many systems simply stop functioning in the case of a system failure and the user is left in the unenviable position of not knowing what has happened. The user will definitely be unsure about whether the activity that resulted in the failure is worth

repeating or not.

Norman [Nor89] argues that in any complex environment — for instance, a new application — one should always expect the unexpected. To deal with the unexpected, Norman concludes that continuous and informative feedback is essential. Norman [Nor86] mentions three different concepts which exist when the human computer interaction process is considered:

1. *The design model* — the model held in the system designer's mind of how the system should work.

2. *The user's model* — the mental model of the system, as built up by the user during user interaction with the system.

3. *The system image* — which portrays the physical structure of the system.

As users use a system, they build up a model of how the system works. In a conversation, the speaker is also able to gauge the knowledge of the listener during the course of a conversation — also building up a mental image of the thought processes and attitudes of the person being addressed. With respect to building this model for computer applications, users tend *not* to read manuals, wanting rather to find out for themselves how the system works [CR87]. They also tend to be impatient to get on with their task and don't want to spend hours being taught how to use a system [Bor91]. This is in accordance with cognitive principles, which advocate "learning by doing" [And83, Man87]. Because of this, the user model will not be based on the design model, but rather on the system image. The designer thus has a difficult task in making this system image explicit, intelligible and constant [Nor86].

Therefore, *feedback* is far superior to user manuals for helping the user to build up a correct internal model. The role of clear explanations in this process is vital [Lew86]. Explanations of system actions can provide a sense of the underlying *purpose* of the system's response to a user's actions. Chan *et al.* [CWS95] have also shown that an active feedback system greatly improves user performance.

This section has discussed the need for feedback from a cognitive perspective. This perspective is vital in understanding the need for feedback with respect to application use when the application has the user's full attention and nothing occurs during use of the application. This is an unrealistic expectation though, since a user's working day will be interspersed with disruptions of all types, which serve to make feedback even more crucial. The following sections consider the timeliness and quality of feedback.

4.4 When must Feedback be Given?

The need for feedback has been argued in the previous section. Different authors have attempted to provide guidelines to help developers to provide the right level of feedback.

Waern [Wae89] suggests that feedback should not be delayed, since the user needs it continuously to support a sequence of mental operations. Other researchers also urge that immediate and continuous feedback be provided [Shn86, App87, Nie93]. Planas and Treurniet [PT88] have shown that continuous feedback reduces annoyance caused by slow responses.

Nielsen contrasts different types of feedback with its persistence. Persistent feedback refers to something such as a disk space or performance indicator, while transient feedback refers to error messages. Others, such as Marshall *et al.* [MNG87] point out the difference between what they refer to as *required* (during execution of the task) versus *confirmatory* (at the end of the task) feedback. The former is required for more complex tasks, while the latter is suitable for simpler tasks or tasks for which the user can be considered to be an expert.

Finally, it has been shown that feedback has an effect on the level to which a particular task is automated[1]. When the feedback is immediately available, the user will be less likely to automate the task and more likely to work in a controlled mode — making less errors. Thus, in complex tasks for which the user needs to concentrate in order to notice exceptional circumstances which will require handling, the feedback should be more intense and available than for simple tasks which can be automated without risk [Gar87]. To summarise, the rule seems to be: "Always provide feedback, for every action, and make sure it is completely unambiguous and informative". Quite a tall order.

4.5 What is Good Feedback?

Some feedback needs are fairly standard, such as the need to alter the display to indicate that something has been selected. Standard requirements such as these lead to uniform treatments in accordance with the *Human-Computer Interaction* (HCI) principles of consistency of interface. For example, text is often highlighted to indicate selection while an icon is inverted to show that it has been selected. However, some feedback needs are not nearly as straightforward and the developer may not have ready guidelines to follow. An excellent example of this is the diverse treatment accorded to error management. Some applications will display an error message which requires some acknowledgement from the user before work can continue. Others simply generate a beep and refuse to continue until the user provides a correct response, and yet others will display an enigmatic message and close the application. It is difficult to provide a general rule about the exact nature of the feedback since it is directly dependent on the nature of the task.

The feedback discussed in previous sections has referred to the communication of the "here and now" state of the system to the user. This feedback model is impoverished and a strong case can be made to motivate the extension of the concept to encompass a historical perspective that would add a dimension to feedback hitherto unexplored.

[1] That is, promoted from the attentional level to the schematic level, at which the user no longer thinks about what they are doing.

It has been noted by various researchers that discourse typically has an incremental quality about it [CM93, LM94]. When people converse they often refer back to some part of their conversation in order to explain their present remarks. Dix [Dix91] argues that it is difficult for users to manage and visualise this "sense of history" in their interaction with the computer, especially since the current interface is based more on recognition than recall. The user has no need to remember lists of commands but simply chooses one from a menu. This historical need was also noted by Tweedie [Twe97] who argues that past input and output should be linked so that all historical input and output relationships can be explored directly. This is echoed by Shneiderman [Shn98]. Often the application's only concession to a user's need for this is the provision of an undo facility. Even where some tutorial or visualisation applications supply the user with a log file containing previous explanations [EL96, DJA93], this does not link the explanations to user actions and is of limited assistance in providing feedback. If Norman's stages of human activity are considered, the explanations only provide step 5 — the system state — whereas the user needs to understand the link between step 4 (their actions) and step 5 in order correctly to interpret the state of the system.

Another look at the conversational model will serve to illustrate this concept. If someone is recounting a conversation with a third party, the structure of the narrative will take the form: "She said ....., and then I said ...". This is so that the person being addressed can understand the context of the narrator's statements. It is no good only hearing one side of the conversation and if the narrator chooses to present only one side it will often lead to the listener being given an incomplete view which is not conducive to understanding.

In addressing the question of which type of feedback is to be provided, it is therefore appropriate to consider the need for the portrayal of previous system states so that the user can refer to it in order to understand the present state of the system. Rich and Sidner [RS97] refer to the need to relate current actions to the global context and interaction history. The previous paragraph has motivated the need to keep a history of both the user's actions, together with the system's response. This type of information could be referred to as *archival feedback*, as opposed to *immediate feedback* which communicates the present state of the system. Such archival feedback provides the facility often used in conversation when a person refers to a previous statement and builds on it. In the light of this discussion, good feedback would thus involve giving the user both immediate and archival feedback.

The previous section discussed the use to which feedback will be put. We can now bring these two concepts together, by marrying the concepts of use of feedback with either immediate or archival feedback, as follows:

1. *signaling a response* — satisfied by immediate feedback;

2. *changing behaviour and promoting understanding* — satisfied by both immediate and archival feedback. Immediate feedback allows the user to judge the immediate state of the system, while archival feedback supports the generation of a deeper understanding

of how the system arrived at that state over a period of time;

4.5.1 Examples of Inadequate or Bad Feedback

Given the wealth of examples available, this section could become long and arduous to read, but instead will consist of a few examples encountered in using widely known applications:

- Ghostscript — Produced this message, which could not be cleared till the system was rebooted. `StartDocPrinter() failed, error code 1722.`
- Internet Explorer — The buttons at the top of the Internet Explorer window include a `Print` button, which provides no feedback. The user will often be unsure about whether the document has been printed or not — especially when a specific train of thought has been interrupted. Since the printer might be in another room, the only way to make sure is to get up and check whether the document has come out of the printer. It would be relatively simple to change the appearance of the button to indicate that the displayed page had been printed. This would leave the user in no doubt about whether the command had been acknowledged by the system and whether the page had been printed or not. This is an example of the reliance of some applications on other tools on the system to provide the required feedback.
- xv — Upon issuing the `Print` command the user hears a series of beeps and nothing comes out of the printer[2].
- White Tiger[3] — In contrast to the paradigm applied by most applications, this application wants the user to specify the location of the output file, before specifying the location of the input file. Doing things the other way round causes the output to be written to the wrong file. No feedback is provided about this and it is often only discovered after a period of frustration.

Whereas the previous examples merely give inadequate or no feedback, it should be noted that bad feedback is worse than no feedback at all. Some examples are:

- Internet Explorer — The message: "`Application error. Press OK to exit, and Cancel to debug.`" is frequently displayed. Why on earth should a user be offered the opportunity of debugging? When the user reluctantly chooses to exit, another useless message is displayed: "`Application Error`", and the user is invited to click on an `OK` button. It then closes down the application, whether the user likes it or not. This is worse than inadequate, it is completely useless!
- Microsoft Outlook Express — If one uses their facility for connecting via a modem to a mail server, the program will sometimes display a message upon disconnecting that looks very like an error message:

[2] I *still* haven't figured this one out!

[3] A shareware application which converts MP3 files to WAV files, among other things.

```
Internet Explorer cannot open the Internet site
http://www.freeserve.com/email/outlook/infopane.htm.
The connection with the server was reset.
```

This leads the novice user to the conclusion that "something has gone wrong", whereas the application is merely informing the user that the modem has been disconnected.

Why is feedback provision so inadequate? It could be because the application programmer is expected to provide for the feedback needs of at least three completely different types of users (end-user, programmer and system support) — often without guidelines, trusting only instinct. Any application will be used by a variety of users during its lifetime. The first is the programmer, the next is the end-user and finally the system-support person supplying assistance to the end-user. Each needs a different type or flavour of feedback. Many applications in use today evidence the variability of feedback provided by different programmers. Some possible reasons for this variability will be briefly discussed:

1. *Lack of Human-Computer Interaction (HCI) training.* The programmer belongs to the world of information technology and finds it hard to conceive of users who do not have this understanding. The system developer brings a store of background knowledge to the task and tends to assume a certain taken-for-granted knowledge in the end-user [FFW88]. Assumptions about the expectations of people not known to the developer are bound to be inaccurate. Consequently, it is extremely difficult, especially for a programmer without formal training in human-computer interaction, to provide feedback at the level required by the user. Since there is a shortage of programmers with specific training in human-computer interaction [MB99, Str99], it is realistic to expect that most applications will fall short of the ideal level of feedback.

2. *Insufficient communication with the user.* There is a very real difficulty in judging the knowledge of the user. The programmer becomes so wrapped up in the program, spending hours and hours developing it, that it is extremely difficult to remember exactly what can be presumed fore-knowledge and what should be imparted to the end-user.

3. *Layering of systems.* Many errors occur at a depth in the system where there is no awareness of the current state of the dialogue with the user. Thus the program from which the reporting emanates typically has no idea of the context from which the user needs to be relocated.

4. *Difference in goals.* Grudin [Gru87] published a paper which looked at the issue of technologies in which one person did work for which another person would reap the benefits. This was coined by Norman [Nor94] as *Grudin's Law*:

   > "When those who benefit are not those who do the work, then the technology is likely to fail, or, at least, be subverted."

The programmer achieves little benefit from providing the right level of reporting for other types of user. Indeed, some organisations actually *profit* from software which provides inadequate feedback — by requiring users to pay for advice on using their systems.

5. *Unrealistic expectations of users' working environment.* The user rarely devotes full attention to any application 100% of the time. Applications seldom take this into account and provide little or no support to users who are frequently interrupted [Jam00].

It is clear from the previous discussion that feedback is vital and that it is often neglected by application developers, to the detriment of end-users. The published guidelines do not seem to go far enough in establishing a clear path for developers to follow in providing the necessary feedback. The following section will attempt to remedy this by consolidating the work by researchers in this field into a list of desirable feedback features.

4.5.2 List of Desirable Feedback Features

It would be useful to have some sort of list of requirements, a milestone to measure actual system feedback against what could or should be provided. Bannon [Ban89] points out the need for research results which have an applicability to design, rather than concentrating on merely delivering tools for post-factum analysis. The beginning of this section argued for the provision of both immediate and archival feedback. The features listed below have been chosen to meet both those needs. A list of desirable features would include the following:

Immediate Feedback

1. Keep the user informed about system state [SKB99], i.e. whether the system [FvD82]:
   - has received their request;
   - is working on it;
   - has a problem; or
   - has completed the task.
2. Explain unusual occurrences and errors. Provide context sensitive assistance [Gar87]. Ensure that it is absolutely clear whether a feedback message is indicating an error or an event of interest which is being reported merely in the interests of good communication.
3. Make visible what would be invisible and improve the user's feeling of control [Nor98]. Give each action an obvious and immediate effect [Shn98]. In addition, the feedback should be structured in such a way that the user is left in no doubt as to which particular action the feedback refers to [Ham87, Gar87], with Nielsen [Nie93] advising

that the user's input should be rephrased and returned to indicate what the system did as a result.

4. Provide a form of feedback which is consistent across applications. The degree of low-level consistency evidenced by windowing systems could usefully be extended to feedback as well. This type of consistency is very comforting to the user.

Archival Feedback

1. Mental aids to help users remember things [Shn98, Nor98]. People have severely limited memories, as illustrated by the following examples [Ols87]:
   - Users sometimes forget what they have done, especially if they are interrupted during a processing session.
   - Users often do not detect their errors. Sometimes the user is vaguely aware that something has gone wrong, but has no idea how this occurred.
   - Difficulties are often experienced in holding recently experienced information until needed.
   - Users experience problems retaining information retrieved from long-term memory — such as remembering where they are in a plan of action.
2. Provide inter-referential feedback. Draper [Dra86] points out the importance of a mutual reference, or *link*, between user input and application reaction so that previous parts of the user-machine dialogue can be referred to.

It is unusual for any system to provide a standard of feedback which copes with these problems. In addition, it seems to be a waste of programmer resources to duplicate some of these functions for each and every application. Furthermore, in providing the feedback, there are difficulties which beset application programmers, as described in the next section.

4.5.3 Provisos

Humans are diverse and wondrous creatures and their very versatility makes the provision of feedback, along with other features of human-computer interaction, anything but straightforward. Shneiderman [Shn98] discusses the following factors which should be kept in mind:

- *Physical abilities and physical workspaces.* Applications often use a beep sound to signal an error, while a mail reading facility running on the same machine will use the same sound to signal the arrival of a message. While the user might not have a problem distinguishing these signals from one another, a noisy working environment could detract from the efficacy of these signals.

- *Cognitive and perceptual abilities.* The following classification of human cognitive processes is given by the *Ergonomics Abstracts* journal:
  - Short-term memory.
  - Long-term memory.
  - Problem solving.
  - Decision making.
  - Attention.
  - Search and scanning.
  - Time perception.

  People also have different cognitive styles [Jac73]. However, Tan and Lo [TL91] find that there is evidence, citing research done in [Hub83, TB80], to suggest that cognitive styles are not a critical factor in user interface design. Cognitive styles will therefore not be considered to affect the provision of feedback.

- *Personality differences.* There are differences in the way people feel about computers. Some like them while others loathe them. Shneiderman argues that there are differences between males and females with respect to computers too, but points out that this difference has yet to be fully explored.

- *Cultural and international diversity.* Examples of concerns for user-interface developers in this category could be left-to-right versus right-to-left, currency differences, addresses or national identification.

- *Disabilities.* Visual feedback is not much use to blind users and deaf users will not be aware of audio feedback.

- *Limitations of elderly users.* With age people find it more difficult to distinguish between colours. Older users are slower to react and can often not read small print on the screen and can hold less information in their working memory at a given time [Gar87].

- *Experience.* There is a difference in performance and in expectations between the user who has had very little computer experience and one who is familiar with computers. The former is very easily intimidated by computer applications and will need far more explanation of basic functions. The experienced user, even if encountering a new application for the first time, is not as easily discouraged and needs less reassurance.

It is difficult, if not impossible, for an application to provide feedback which is customisable to the needs of a specific user as shown above.

4.5.4 Differing User Roles

Any computer application has different types of users during successive stages of the life cycle of the application. At least three distinct categories can be identified, as differentiated by their different roles. The first user is the *application programmer*, who will be creating the end-user application. The next is the *end-user*, the client for whom the application has been created. The third is the *system-support person* responsible for providing technical assistance and error intervention to end-users. Each type of user has very different feedback needs:

1. The *application programmer* will need highly technical feedback. The goal of the programmer is to produce a working application and the feedback provided must therefore assist in the debugging process. The type of feedback required could be the parameters provided in a particular method call or the return value supplied or a stack trace of an exception thrown by a method call.

2. The *end-user* needs to be given feedback relating to specific goals, linked directly to the task being carried out. The feedback must be on a much higher level than that required by the programmer.

3. The *system-support* staff will often be summoned when the end-user has received a message from an application which is indecipherable, or due to an error message indicating some sort of problem. The first question asked by system support staff will be: "What were you doing?" followed by, "What message did the system display?". This will assist them in tracking down the source of the problem.

The application programmer is expected to provide for the feedback needs of these three completely different types of users. It is extremely difficult for an application to provide for *all* these different user needs and many applications in use today are evidence of the variability of this provision by different programmers.

4.6 Feedback Format

The previous sections have argued the necessity of feedback and discussed the type of feedback to be provided as well as the difficulties inherent in feedback provision have been discussed. This section will address the issue of how feedback should be provided in a visual format.

4.6.1 Textual versus Graphical Feedback

The first issue to be resolved is whether feedback should be given in textual or graphical format. In human discourse, many different communication channels are used to provide

feedback. Apart from utterances, people also use gestures, gaze and body stance to communicate their understanding of what is being said [EH93]. A feedback model based only on textual descriptions will therefore not exploit the multiple possibilities available in providing feedback to the user.

In conveying a message it is often useful to make judicious use of metaphor, perhaps involving graphical components which can be superior to a purely textual description [DFAB93]. An example of this is the use of the spreadsheet metaphor for accounting applications. A well chosen metaphor is invaluable in increasing an end-user's familiarity with an application. Metaphor must be used with caution, though, since an incorrect choice could make things even more confusing for the user.

Shneiderman advises that a feedback display should be consistent — using the same colours, formats, captialisation etc. so that users will know what to expect, and that feedback should always be given where it is easily detected [Shn98]. This can apply equally to textual or graphical feedback. However, there is a body of research which points unhesitatingly towards the advisability of graphical feedback.

Norman advises that sound and graphics should be investigated [Nor98]. Faulkner, too, advises that feedback be presented in a graphical format and that all feedback messages should be clear and unequivocal [Fau98]. Phillips [Phi86] argues that visual imagery is superior to verbal representation in aiding memory and thinking. Gardiner [Gar87] agrees, saying that recall is better for dynamically interacting items than for items stored in isolation. She avers that recall is further improved if items are presented pictorially, rather than textually.

From a cognitive point of view, graphical feedback may be far more helpful, since users have particular strengths which can be utilised by non-textual feedback mechanisms such as processing visual information rapidly, coordinating multiple sources of information and making inferences about concepts or rules from past experiences [Ols87].

Since the user's interaction with modern computer systems is essentially based on recognition, rather than recall, and is intensely visual, it would be less than optimal to try to describe the actions in a textual format. The representation chosen for a particular set of data will indeed make a difference [Sim69] — some representations allowing users to perceive one type of pattern in the data, others revealing something totally different. We should therefore explore possibilities for portraying feedback in a graphical format. In order to do this, there must be a visualisation of the information that we are attempting to portray — a graphical format which will be assimilated by the user more easily than a text description. The following section will define the concept of visualisation and Section 4.6.3 will briefly address issues that must be borne in mind in deciding on a visualisation.

4.6.2 What Does Visualisation Do?

Visualisation provides an interface between the human mind and the computer. In visualising information, the challenge is to find designs that reveal detail and complexity, rather than presenting the user with a confusing profusion of clutter. The failure of the design will sometimes be blamed on the complexity of the data, or on lack of understanding on the part of the viewer [Tuf90]. Chen [Che99] explains that information visualisation is composed of two essential activities: structural modelling and visual representation. Once a visualisation structure has been identified, the mechanisms and design techniques must be chosen to present a visualisation of the information. Shneiderman [Shn98] cites the following tasks that need to be supported by a visualisation:

- Overview — to gain an overview of the whole collection;
- Zoom — zoom in on items of interest;
- Filter — filter out non-interesting items;
- Details-on-demand — select an item and get more information about it;
- Relate — view relationships between items;
- History — keep a history of actions to support undo or replay; and
- Extract — allow the user to extract subsets of the information.

In choosing a visualisation, a designer has to work at different levels. The first, low-level choice is concerned with the visual variables available — such as size, colour, shape and symbols. The second far more difficult choice pertains to the use to which these visual features will be put in order to present the required information. Dix [Dix91] notes the difficulty of choosing a particular technique for some data set. Directives in choosing techniques are discussed in Section 4.6.3.

4.6.3 Restrictions

There are some guidelines to be borne in mind when visualising information [Cha99a, Tuf90]:

- Do not overload the user with information. Rather provide tools which will allow the user to get extra information.
- Gershon *et al.* [GEC98] urge that visualisation systems should be based on human capabilities of perception and information processing.
- The layering of information is difficult. Tufte [Tuf90] advises the importance of a proper relationship among information layers. Information can be separated by using colour, shape, size or value (light to dark). Separation is sometimes achieved by means of a grid — the grid should not dominate, but should be muted relative to the data.

- Small multiple designs which visually represent comparisons of change are the best way to answer questions about quantities. Small multiples reveal a range of options. Tufte warns that comparisons should be enforced within the scope of the eyespan. There is also a continuous trade-off between the maintenance of context and the provision of visualisation to support comparison.

- Visualising time and space involves the design of maps and time-series. Examples of this type of visualisation include road maps, itinerary design and timetables. Many of these depict changes in both time and space. A novel application of this technique has been applied to the visualisation of dance routines.

Layout is important and Chen [Che99] emphasises this, pointing out that a good layout conveys the key features of the system to a wide range of users, while a poor layout would obscure them. Vanderdonckt and Gillo [VG94] give five sets of visual techniques which can be used as guidelines for presenting a layout:

- physical — balance, symmetry, regularity, alignment, proportion and horizontality;
- composition — simplicity, economy, understatement, neutrality, singularity, positivity and transparency;
- association — unity, repartition, grouping and sparing;
- ordering — consistency, predictability, sequentiality and continuity;
- photographic — sharpness, roundness, stability, leveling, activeness, subtlety, representation, realism and flatness.

Vanderdonckt and Gillo emphasise that these techniques cannot all be applied to every situation, but that others are always to be applied. Which techniques should be applied is completely dependent on the nature of the data being displayed. Throughout this process, we should keep in mind that we are seeking to reduce the complexity of the data and allow the user to use information which, if presented badly, will be useless.

The question of quirks was fully explored in Chapter 3. The following section will consider the role that feedback can play in alleviating the negative aspects of quirks and in assisting the user in dealing with them.

4.7 Feedback for Quirks

Jambon [Jam96] urges system developers to design with interruptions and errors in mind. He argues that this would decrease the possibility of operators forgetting something critical after handling a quirk, thereby causing a serious accident. The focus of Jambon's research was interfaces for pilots. Errors made by users using other systems may not have such serious repercussions as those made by pilots, but that does not make them any less annoying. The

contribution made by feedback to alleviating the effects of each of the quirk categories will be discussed in the following sections.

4.7.1 Breakdowns

Immediate feedback is not much use if the end-user computer breaks down. Archival feedback can only be useful if it persists. If another part of the distributed system breaks down, it will depend on the forethought of the application designer whether the system will respond in a helpful way or not. If the breakdown was not anticipated by the designer during system development, the user is sure to receive an unintelligible response. Archival feedback could be helpful to the specialist summoned to track the events leading to the breakdown. What *will* be useful is some way of understanding exactly what the problem is together with some indication of the course of action to be taken.

4.7.2 Human Error

The recommendations given for error recovery by Rizzo *et al.* [RPMB96] for supporting the handling of human errors as discussed in Section 3.5.5 will be reiterated here:

1. Make the action perceptible.
2. Display the error message at a high level.
3. Provide an activity log.
4. Allow comparisons.
5. Make the action result available to user evaluation.
6. Provide result explanations.

These are remarkably similar to the desirable feedback features given in Section 4.5.2. The first, second, fifth and sixth recommendations are satisfied by immediate feedback, while the third and fourth are satisfied by archival feedback.

4.7.3 Interruptions

If we consider the stages of activity defined by Norman, and enumerated in Section 4.3, Miyata and Norman [MN86] suggest that an interruption would be least disruptive if it occurred between the evaluation stage and the formation of a new goal or intention. Generally an interruption when the memory load is high is very disruptive, whereas an interruption when the memory load is low — where much of the context is available via external cues — is not as disruptive. Miyata and Norman conclude that interruptions would be most disruptive at the planning and evaluation stages.

It would appear that the great problem with handling of interruptions is that it is often difficult to re-establish context so that the user, in choosing the task to be resumed, has often forgotten all about the stage of important work in progress. Thus, it is clear that the nature of human episodic memory is relevant to the interrupt handling process. Gardiner [Gar87], presents the following facts in her discussion of episodic memory:

1. People have a very limited ability to remember the detailed appearance of novel, visual abstract patterns, even over intervals of a few seconds.
2. Immediate memory is poorer the more complex the visual pattern.
3. Immediate memory for visual abstract patterns is disrupted by even a small amount of distraction.
4. So long as the context in which information is retrieved approximates the context in which it was stored, recognising an item in memory is easier and more efficient than having to recall the item unaided.

Point number 4 is especially relevant to error reporting and interruptions, since it underlies the need to remind users of what they did, in the same format in which it was done, in order to provide context-sensitive assistance — hence once again a motivation for archival feedback. Jambon [Jam00] points out that at design time it is important to have a table of all possible interrupted tasks by all possible interruptions. He advises that for each entry of the table the developer must indicate the context and find out how this context may be stored during the interruption (by the human working memory and/or the interface). The interface can be said to tolerate interruption if, in each case, the programmer can prove that the context may be saved.

4.7.4 Conclusion

There is a commonality in the user's handling of errors, breakdowns and interruptions. In the case of error, the user has to *understand* the cause of the error and *understand* how to recover from it. In the case of breakdowns, the user needs to *understand* what caused the breakdown and what, if any, action should be taken to recover. In the case of interruptions, the user attempting to resume context must *correctly perceive* the state of the application in order to take up their task at the point of interruption.

We can conclude that feedback which enhances the user's comprehension of the application state, and the events that led to that state, is a valuable tool in ensuring that users are able to handle quirks easily. Furthermore, this will comprise a judicious mixture of immediate and archival feedback.

4.8 Summary

This chapter has argued the necessity of feedback and given guidelines about how it should be provided. The need for both immediate and archival feedback has been argued and directives for providing feedback have been given.

It has been pointed out that feedback should be tailored towards the needs of the end-user and it would be a difficult task for applications to provide for all the possibilities mentioned in Section 4.5.2. Thus we can conclude that the provision of feedback is not easily achieved.

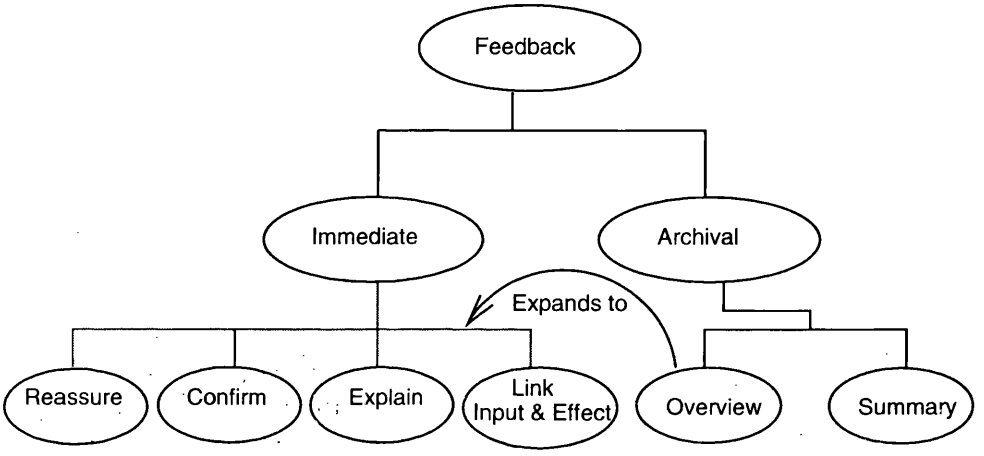

Figure 4.1: A Classification of Feedback

In order to synthesise the recommendations cited in this section, a classification of the nature of feedback has been constructed and is illustrated in Figure 4.1. Feedback should have both immediate and archival features. The immediate feedback should confirm user actions, reassure users that the system is functioning correctly and explain errors if they occur. This should be done in the framework of a reference between the users actions and the resulting system response. The archival feedback should offer an overview of session activities, as well as a summary function so that the user can get a broad view of activities. The overview should offer the facility for the user to choose a particular previous action so that the previous immediate feedback can be duplicated — with the explanatory and confirmatory functions still being useful at that stage.

This chapter concludes the background literature survey. The following section of this dissertation will pose the problem being addressed by this research, propose a solution and cite related work which supports the proposal.

part III

Addressing Feedback Needs in Component-Based Systems

He who joyfully marches in rank and file has already earned my contempt.
He has been given a large brain by mistake,
since for him the spinal cord would suffice.

Albert Einstein

"This affair must be unravelled from within."
He [Hercule Poirot] tapped his forehead.
"These little grey cells. It is 'up to them' ".
Agatha Christie
The Mysterious Affair at Styles. (1920) ch. 10

chapter 5

Problem Description and Proposed Solution

Part I discussed the emergence of component-based systems as a dominant force in software construction, and the importance of feedback to the usability of software. Part II brings these two aspects together to discuss how to support feedback in component-based applications. The diffuse nature of such applications is certain to affect the way in which feedback can be programmed, and the difficulty of recovery from quirks. The solution that is adopted centres around the use of application tracking — i.e. monitoring what the user is doing and how the application is reacting.

Section 5.1 presents a synopsis of the problem being addressed, with special emphasis on the feedback needs of component-based systems. Section 5.2 proposes a solution to the problem. Sections 5.3 and 5.4 discuss the techniques used in the proposed solution, and Section 5.5 justifies the need for the provision of feedback by means of a visualisation of application activity. Section 5.6 consolidates the chapter by providing an outline of some of the benefits and limitations of this approach and summarising the chapter.

5.1 The Problem

Chapter 3 looked at the various problems which interfere with the straighforward use of applications, all of which make feedback vital. Chapter 4 examined the nature of feedback and concluded that the feedback provided to end-users is often woefully inadequate. The research described in this dissertation concentrates on feedback needs in component-based systems. Section 5.1.1 will look at the ways in which feedback has traditionally been provided. Section 5.1.2 will examine reasons for the difficulty in feedback provision in component-based systems (CBSs). Section 5.1.3 will discuss why error recovery becomes more difficult in CBSs, with specific reference to e-commerce systems. Section 5.1.4 concludes this section.

5.1.1 Traditional Ways of Providing Feedback

There are various ways in which we can ensure that adequate feedback is provided. The following sections will discuss different approaches.

5.1.1.1 Guidelines for Programmers

Provision of feedback during application development is often left to the individual programmer's discretion. However, good user-interface design is more than just common sense [Tul93].

There have been some attempts to set out guidelines for many aspects of user-interface design and feedback has not been neglected. Some examples were given in Section 4.4. Thimbleby [Thi90] points out that developers are sceptical of guidelines because of a perception that they are either trivial or difficult to implement, or both. Guidelines are often user- or application-dependent, which makes formulating or following them almost impossible. Thimbelby also notes that adherence to guidelines by no means guarantees that a chosen means of feedback will work, until it is in use. By then, it is probably too late to improve on it.

Expecting programmers to follow guidelines is simply not realistic, as evidenced by the many systems in use today with appalling standards of feedback. Norman and Thomas [NT91] give some examples of problems experienced by users making use of systems which do not give an appropriate response to their actions. Provision of feedback is such a complex activity that it is doubtful that any set of guidelines will ever fit the bill. As in other complex human activities, those who do it well will have difficulty in formulating their methodology.

Furthermore, it is wasteful to have programmers duplicate the coding required to provide identical, non-application specific feedback functionality in application programs. While application specific feedback cannot be replaced by any other facility, many of the extra features discussed in Chapter 4 are almost generic in their nature and offer the possibility of being provided for by a generic facility. For example, many user interfaces provide an explanatory balloon which pops up when the user lingers over some button on the screen.

This facility, which has become almost generic, has so obviously been helpful to the end user that it has become ubiquitous.

5.1.1.2 Comprehensive Online Manuals

This approach is followed with different measures of success by various applications. The literature on online manuals is well established [DPM92, Kea88]. Innovative approaches, such as providing animated help, have also been developed [Thi93a]. In a CBS, however, the late composition of the system makes the development of a comprehensive online manual difficult, if not impossible. There is also currently no standard to which components can satisfy which requires that they provide an online help facility for their component. Even if provided, the diversity of the different component producers would not facilitate a coherent, understandable help facility. Even if the application programmer were to choose to provide a help option, that would not supply the level of feedback that the user needs, but only fulfill an explanatory function. The issue of dynamic feedback is not catered for by online help. This problem is exacerbated by the international nature of most component-based systems.

5.1.1.3 A Feedback Application Programmer Interface

This approach would provide an API, which could be used by a programmer to provide feedback to the end-user. The API might display feedback in a standard window, or be added systematically to the active window in some way. This would become visible whenever the user needed to be apprised of some event, or to denote closure of an action. This option suffers from the same problems as the first, since use of the API is dependent on the vagaries of the individual programmer. It also limits the type of feedback which can be provided.

5.1.1.4 Summary

If we judge the process by its end-product, we can conclude that the traditional ways of providing feedback are not effective. This section has addressed the difficulties of feedback provision in general terms. The following section will discuss the special problems of feedback provision in component-based systems.

5.1.2 Why Feedback Provision is (Even More) Difficult in Component-Based Systems

With respect to traditional usability needs, CBSs are no different, but in CBSs user needs are less likely to be addressed comprehensively. Component-based systems are constructed using components harvested from possibly (and indeed probably) many different sources. The developers of different components will not have met each other, let alone discussed their error reporting and handling mechanisms. This means that each component will handle

errors differently, according to the particular developer's own preferences. The components will also probably have different input conventions, for example, increasing the likelihood of mistakes. Even if components are documented correctly, there will probably still be problems, due to the black-box component approach [BW97].

Consequently, the user-interface developer will not have developed the server components, and is simply given an interface, an API and some component documentation for each server component. Using only these resources, the developer creates a user interface. This developer will probably not anticipate all the errors which could result from the use of each component and not make provision for all of them. This will cause great problems for the user when something goes wrong later on.

Furthermore, the developers of components, CBSs and user interfaces for these systems are mostly people who have a high technical expertise and this can make them unrealistic about the abilities of their end-users. (This makes it very difficult for developers to conceive of a user who has not attained the basic level of technical knowledge that they take for granted.) One only has to read a few papers about CBSs to be convinced of this. Norman [Nor98] puts it very succinctly by stating that "there is no return to innocence". Applications cannot be produced for individual users, since this is not economically viable. Thus the application developer must produce applications for a kind of "generic" end-user and make assumptions about the users' knowledge. A large part of the problem is that there is a basic mismatch of assumptions and knowledge. This gulf has to be bridged effectively if feedback needs are to be met.

Distribution, once again, makes things more complex. CBSs are often distributed over many sites. This adds to the possibility that some parts of the system will not always be available. Such is the nature of distributed systems [Bac97, Mul93]. Users will often be puzzled by such absences and need to be apprised of the reasons for them.

Finally, as we know, feedback is traditionally provided from within the application code, but this approach is flawed because programmers are seldom trained with the HCI skills to provide adequate feedback and it is almost impossible to augment the feedback once the application has been delivered. Furthermore, users functioning in different roles often have completely different feedback needs and it is difficult for an application to provide for all of them adequately.

5.1.3 Why Error Recovery is (Even More) Difficult in Component-Based Systems

Section 3.5 examined error in some detail. There is ample evidence to lead to the conclusion that humans do indeed err, that they are unrealistic about their propensity for making errors and their ability to detect them, and thus, having erred, will convince themselves, in spite of clear evidence to the contrary, that they did not err.

In the days of batch processing, the traditional transaction concept protected databases

from the effects of errors. The application program would start the transaction, make the changes, and either commit or abort the transaction. A human agent would supply the data being used to make the updates, and data entry professionals being highly skilled, made relatively few mistakes. In those days, much use was made of manual checking, with supervisors being responsible for keeping the occurrence of errors down to a minimum.

These days, things are somewhat different. Contributing factors are both the nature of components, as expounded in Chapter 2, and the architecture of these systems, which decrees that the user interface is essentially thin, with much of the logic being encapsulated in the middle tier. Instead of disciplined data-entry specialists exclusively entering data, just about anyone is involved in entering the data to be used in database transactions. Each user of internet e-commerce can, and will, enter data which will make changes to some underlying data store. Very few of these people will be skilled in data entry and we can therefore expect that many errors will be made. These errors will possibly be unlike the fatigue-induced errors generated by data-typists, but regardless of their cause, they can be expected to be far more numerous.

Another factor to be considered is the fact that most users of thin-client e-commerce systems will not have been trained in their use. Since the user will not have been trained to use the system, the user interface will have to be designed with great care. The user must be able to discover everything about the system, based on the perceptible system state. Users are no longer given extensive training in the use of particular systems, this *must* have a significant effect on the way that systems should be designed. The designer of the user interface must be sure to bestow rational behaviour on the application — ensuring that the application behaves in a way that is reasonable and intelligible. When a user makes use of an application, the application must give the impression of being being responsive to user actions, in the same way as humans are responsive to other humans' actions [Suc87]. This, once again, brings us back to Norman's assertion that things invisible should be made visible [Nor98], so that the user can understand the motivation behind the system's actions — as being directly in response to their own actions. While humans routinely correct mistakes made by other humans, it is important to make the distinction between humans and computers. Computer applications are quite simply unable to make these corrections and it would be unwise to contemplate complete reliance on such a scheme. What the application must do is give the user the maximum opportunity of detecting their errors so that they can be corrected.

If this admonition is ignored, it could cause an unwary user to precipitate all sorts of havoc by using a system incorrectly. A simple order form, currently used with great success by catalogue firms, could be a disaster on an e-commerce system. The user could easily enter an item code into the quantity box, for example, and inadvertently order hundreds of items they did not require. The removal of the intelligent human agent, which in manual systems would filter out this type of error, means that even greater care must be exercised to ensure that user errors do not cause disasters.

It is also essential that the user understands when their action (perhaps the click of a button) will cause a transaction to become final. In the days of batch processing, computer applications proceeded from instruction to instruction within predictable time boundaries. So, if the application program did not crash, the program would start a transaction, make some changes to the data and commit or abort the transaction. When a human enters the process, it is no longer possible to allow the program to start a transaction, then wait for the user to enter some data, and then only commit or abort the transaction. Compared to computers, humans are extremely slow and laborious and it is not possible to keep database locks for extended time periods while the user decides which displayed item to choose. The implications of this — what is sometimes called the lazy client problem — is that the program will collect the data from the user, and then start a transaction, make the changes, and commit the transaction. The user will have no control over whether the transaction commits or aborts — due to not being consulted. It will probably happen automatically, as described in Chapter 2.

For example, the online bookseller site, `www.amazon.co.uk`, has obviously been designed with great forethought. The user is continuously told, throughout the ordering process, that nothing is final until the last screen has been reached and a confirmation has been obtained. When users reach the confirmation screen, and confirm the transaction, they are left in no doubt about exactly what they have ordered, that the transaction has been accepted and the order placed. To ensure that this is understood, email is dispatched immediately, further reinforcing the sense of closure.

The online flight reservation site, `expedia.co.uk`, has been less well designed. A user wanting to book a flight uses a search process to choose a flight from a displayed list. The user then has the choice of reserving the flight for 24 hours or purchasing it directly. In either case, the user is asked to enter credit card details and all personal data. Once this has been entered, and the user clicks on the "reserve" button, they rightly expect that the reservation has been made, or the tickets purchased. They hope in vain. Only at this stage does the system make contact with the airline's computers, to ensure that the flight is available. I once entered my details no less than six times with this particular site, each time being informed that the flight was full, before I wrote it off in disgust and used a travel agent instead.

If, having completed a transaction, the user realising he or she has made a mistake, will often find it extremely difficult to correct the mistake. Traditional monolithic systems provide an undo facility, so that one can back up to a previous state, thus undoing the error [LN86]. This is very useful for most applications. However, in transactional CBSs, undo is unlikely to be an option. It will probably not be possible for the application to offer an undo facility. In CBSs where the user-interface program communicates with an interception-based component-oriented middleware layer, each method call is potentially a complete transaction, so an erroneous action which succeeds probably results in a transaction being committed. Other transactions might already have used the data resulting from that

transaction.

The only option for CBSs is for a compensating transaction to be executed. So, a user using an e-commerce system selling gardening products to place an order could incorrectly order 11 garden gnomes (by pressing the "1" key too hard), instead of only one. To correct this, a compensating transaction, cancelling the order of 10 gnomes, would need to be executed.

Dix *et al.* [DFAB93] refer to the concept of *forward recovery*, as opposed to *backward recovery* (undo). Jambon [Jam98], in his taxonomy of error recovery, discusses the different states a system could be in after the occurrence of an error. He emphasises the fact that the state arrived at after forward recovery is not necessarily the same as the state arrived at after normal execution. In the same way, recovery after a crash will leave the system in a state which is not equal to the initial state.

The amazon site offers the user the opportunity of executing a compensating transaction via email, or telephone. This would be an additional transaction, since the original one would have been processed already. It is essential that system developers bear these issues in mind while developing their system.

The state of the system — your credit card account, your temper and the space taken in your garden by your acquisition — will not be the same after backward recovery, as after forward recovery. For example:

- *Backward recovery* — the user enters a quantity of 11, instead of 1. Before clicking on the confirmation button, the error is noticed and corrected. The user confirms the transaction by means of some confirmatory gesture such as clicking on a button. The result: one garden gnome arrives and the credit card account is debited with 20 pounds.

- *Forward recovery* — the user enters a quantity of 11, instead of 1. This is not noticed, and the order is placed. The credit card is debited in the amount of 220 pounds. The error is discovered:

  - *before* the gnomes are delivered. The user contacts the organisaton and cancels the order. Result: a compensating transaction goes through, cancelling the order for 10 gnomes and crediting the credit card account in the amount of 200 pounds. It may seem that the end result is the same. Perhaps it will be, but there is a bigger picture. Suppose the user only realised the error a day after the order was placed. What if the user tried to purchase another item and could not because the credit card limit had been reached? What if the credit card account was printed, and interest payable calculated, during those 24 hours? Either way the user is in for a nasty surprise.
  - *after* the gnomes are delivered. If the organisation is customer-centred, it may take the excess gnomes back without charging extra for collecting them again.

> It will probably be a much bigger job getting the forward recovery done in this case and the effects on the user's temper will be considerable. That convenient scapegoat, "the computer", will probably be blamed for the error and the user might be reluctant to order online again.

The previous discussion merely underlines the need for great care to be exercised when designing these systems — so that the user is given every opportunity to realise that an error has been made, facilitating rapid and painless backward recovery. Should an error be undetected, the system can make life much simpler by making the user's forward recovery process as painless as possible too.

5.1.4 Conclusion

Feedback can be considered to be "making visible that which should be visible, and hiding what is irrelevant" [Nor98]. This is not merely a matter of common sense, as is abundantly obvious to any user of computer applications, but rather an issue which should be given due consideration. It is clear that research into mechanisms and guidelines for the provision of feedback are in an unresolved state, so that many programmers currently are left with no choice but to depend on their own common sense.

5.2 The Proposed Solution

The previous section concluded that the manner of providing feedback, and standards for ensuring the quality thereof, are an open question. Feedback must, at present, be provided during the development of an application front-end[1], and it is extremely difficult to remedy applications which provide inadequate feedback, once they are in use.

Dynamic feedback and error reporting must also presently be provided by the programmer in addition to all the other work. In assisting the programmer to improve the level of feedback provided, there are three prime tenets:

1. It is necessary to make the programmer's task simpler. The traditional approaches to providing better feedback — training programmers and providing guidelines — are doomed to failure since they require an extra measure of effort on the part of the programmer. Chapter 4 explored this issue and concluded that the programmer has an enormous task in satisfying a myriad of requirements, during implementation of an application. On top of that there are ever-present deadlines and inevitable technical problems. It makes no sense to add to this load. Therefore any proposed solution should have as its first tenet the reduction and easing of the programmer's workload.

2. Inseparable from the previous tenet, is the need to provide feedback independently of the application. It is counter-productive to expect the programmer to change program-

[1] The rest of this chapter will refer to *front-end applications* simply as *applications*

ming style to suit any new methodology. If the programmer has to make function calls to facilitate extra feedback, it is not likely to be successful. Thus the mechanism chosen to facilitate extra feedback should be as independent as possible of the application program, and be easy to understand and use.

This leads to the logical conclusion that we should consider feedback to be of two types: *application-internal* and *application-external*. Application-internal feedback will respond to user inputs which do not require the application to interact or communicate with any external entity. This type of feedback will convey information about internal application functioning such as, for example, reporting a subtotal, or registering receipt of a user-interface customisation instruction. Application-external feedback is required when the application interacts with the environment, the user, and the rest of the CBS. This split is made so that we can argue that different feedback needs must be handled in distinctly different ways:

- *Application-internal feedback* should be provided by the application programmer who is completely in touch with the inner functioning of the application.
- *Application-external feedback*, on the other hand, can be provided in a generic manner for all applications, since the applications are necessarily communicating with external entities, so that applications could all fall foul of exactly the same types of errors. Furthermore, each component-based framework includes a generic architecture which can be exploited to build a generic feedback mechanism using application tracking.

Another perspective could consider *component-based* versus *non-component-based* feedback. Any component-based interaction will necessarily entail communication with other components, whereas non-component-based activity could be executed entirely within the application itself.

It is reasonable to assume that there is a benefit to be derived from providing these two types of feedback needs in different ways.

3. Chapter 4 drew the conclusion that feedback should be both immediate and archival, and that it should be supplied in a visual or graphical format, rather than providing solely textual feedback. Any tool which is provided for augmenting feedback should therefore give due consideration to offering feedback in as visual a format as possible.

   This doesn't mean that all feedback should be iconic rather than textual. Textual abstractions have been developed over the past 4000 years, and are often far more effective than pictures. Thus total reliance on text, or total reliance on pictures, will never suffice. Feedback should be tailored according to the data being displayed, and the user's needs.

After consideration of these tenets, and contemplation of various established techniques, the proposed solution treats the provision of application-external feedback as a *separate concern*, which can be dealt with independently of the basic functionality of the program.

Unlike many other tools which provide for the separation of behavioural from functional concerns, the approach applied here is that functionality should be catered for with minimal participation by the programmer — by providing the feedback independently of the application. The use of the separation of concerns technique satisfies the first tenet. Making use of application tracking to obtain the required information to provide the feedback satisfies the second tenet. The third tenet will be satisfied by investigating techniques for visualisation of application activity.

The following sections will take a look at the research into the areas of separation of concerns (Section 5.3) and application tracking (Section 5.4). Section 5.5 will address issues pertaining to the visualisation of the information thus obtained.

5.3 First Mechanism — Separation of Concerns

Programmers have to deal with a considerable amount of complexity — this being inherent in their task. They have to deal not only with the programming of the required functionality, but also with other important issues like replication of components, distribution, real-time configuration, synchronisation and persistence. Wherever possible, software development systems should isolate the various aspects so as to help the programmer focus on specific tasks. Approaches to this vary from separating the specification of concerns — which implies that the programmer can implement the functionality separately — to proposed orthogonality of the specific issue, which implies that the work is done on *behalf* of the programmer, without any effort on their part.

5.3.1 Separate Specification of Concerns

Some research has been done into providing programmers with tools which separate the behavioural features of the software from the functional features [GGM97]. Kiczales [Kic96] introduces *aspect-oriented programming.* Aspects could refer to location, communication and synchronisation, and once specified, they can be automatically combined with the application program by using some tool, such as AspectJ[Asp98], to arrive at the executable application. For example, Kersten and Murphy [KM99] built a web-based learning environment and used aspects to support its runtime configuration. Some examples illustrate the separate concerns approach with respect to [HL95]:

- *Process synchronisation* — in which details about the interaction of concurrently executing processes have been separated from those processes. Frolund and Agha [FA93] have developed language support which enables multi-object coordination. The coordination patterns are specified abstractly, in the form of constraints, which control

the invocation of a specified group of objects. Properties such as ordering and atomicity are enforced by means of these constraints.

Lopes and Lieberherr [LL94] describe an approach to concurrent object-oriented programming which separates synchronisation schemes from the basic behaviour of the application. They introduce a new level of abstraction, called the adaptive level, which describes concurrency control requirements. By using the original program and the adaptive constructs, a complete and correct application program is generated.

- *Location control* — The AL-1/D system [OI94] allows dynamic object location control using meta-level programming. Okamura and Ishikawa make use of computational reflection and a meta-level architecture to separate the programmer-defined computational algorithm from the location control mechanism. This allows programmers to control object location more flexibly than with traditional approaches.

- *Real-time constraints* — Aksit *et al.* [ABvdSB94] make use of composition filters to effectively separate the real-time concerns from other method concerns. Their composition filters are used to allow messages between objects to carry timing information, which allows the receiver of the messages to take the sender's timing constraint into account. These filters catch and affect the real-time properties of messages interacting with an object. Aksit *et al.* argue that the considered configuration of their filters can be used to specify real-time constraints.

- *Distribution* — Since the failure semantics for distributed systems is obviously different from centralised systems, separating this distribution concern is not simple. Stroud [Str93] points out that it can only be done successfully if centralised semantics are considered to be a special case of distributed semantics. Guerraoui [GGM97] describes Garf, a software development tool which provides a library of abstractions to simplify distributed programming. Garf encourages programmers to develop application components by focusing initially only on their functionality. Then, without changing these components, the distribution and replication features can be activated.

  Another approach, for Java, is the *Kan* project (`www.cs.ucsb.edu/~dsl/Kan`) [Jam99a]. Kan provides extensions to the Java language which allows the programmer to mark classes of objects as distributed objects so that the runtime system then manages the distribution, replication and migration of these classes. It either creates distributed objects on specified machines or instructs Kan to choose locations. The Kan system can be used to adapt a Java program to run on multiple machines.

- *User-interface code* — the Chiron-1 User Interface development system [TJ93] introduces a series of layers that separate the user-interface code from the application code by using user-interface agents called *artists* which are attached to abstract data types. Operations on the abstract data types automatically trigger user-interface activities.

The Teallach model-based approach [BMP+99] allows the application developer to specify task and presentation requirements independently from the database contents. The domain model, which reflects the database structure, is meshed together with the other models and the application program is generated automatically.

- *User manuals* — Thimbleby [Thi93b] developed Hyperdoc, a system which allowed a programmer to develop a user manual alongside the user interface, so that the user manual mirrors the structure of the user interface. The programmer can add to either the user manual or the program, and the matching section in either the program or the manual will be modified automatically.

- *Exception handling* — Dellarocas [Del98] makes a case for separating exception handling from normal system operation. An exception handling service is provided for use by component developers, which uses a knowledge base to describe the failure of the system to the user.

5.3.2 Orthogonality of Concerns

This approach is rather different from the previous one. Orthogonality frees the user from the concern altogether. This means that the issue will be taken care of by the underlying system. Examples are far more difficult to find, and include:

1. *Persistence* — orthogonality of persistence was proposed by Atkinson and Morrison [AM95] where the programmer simply identifies a persistent root, and ensures that all persistent objects are reachable from this root. The approach is demonstrated by the development of the persistent programming language PJama [PAD+97].

2. *Location* — COM and CORBA illustrate an orthogonal approach to component location. The programmer never has to be concerned with the location of the server component being used — these details are taken care of by the underlying component communication architecture.

5.3.3 Summary

It is clear from this list, which is by no means exhaustive, that there are many methods of reducing complexity in application development. Most of the examples shown above have required the developer to program both the basic and special concerns, albeit separately. Kiczales [Kic96] argues that this helps to reduce the complexity with which the developer has to cope. Others, such as orthogonal persistence, do most of the work for the programmer in a transparent fashion. The author is not arguing orthogonality of feedback, but rather approaching the problem by providing a tool which will help the programmer to provide the feedback as easily as possible.

5.4 Second Mechanism — Application Tracking

Application tracking is a generic term that refers to the observation of some aspect of an application for one or more purposes. For example, the development of an application throughout its lifecycle could be traced in order to assist management in controlling a software project. On the other hand, a workflow application could be "tracked" to ascertain its route through the intranet, and to ensure that it had not crashed before completing its task. Yet another type of tracking could concentrate on the licencing perspective — with software vendors licencing their product for a specific time period, and installing a licence server on the client's system to ensure that the products being used are indeed licenced.

The meaning attributed to "application tracking" in this dissertation will be the *observation of the behaviour* of an application during its execution. Within these boundaries many motivations for tracking exist. Among these could be a need to:

- understand the application execution process, especially if the application is distributed or runs on parallel processors;
- provide information needed to carry out system tuning;
- satisfy security requirements;
- support debugging purposes;
- provide an audit trail; or
- provide extra support for end-users.

Application tracking is often achieved by adding extra code to the fundamental application code. The execution of this code observes and reports on the behaviour required. An important aspect of application tracking is how this extra code is added. It must not intefere with the normal running of the application, and yet achieve its goals. This section describes different approaches to adding application tracking.

However, before considering this aspect of tracking, it is necessary to discuss the focus or reason for the tracking activity. Applications can be tracked from at least two different *perspectives*, either tracking the user interaction with the application or watching application interaction with the rest of the system. Each will be discussed in the following two sections, followed by a discussion which focuses on the actual insertion of the code to facilitate tracking in Sections 5.4.3 and 5.4.4.

5.4.1 First Perspective — User-Interface Tracking

This type of tracking has an interest in the user's interaction with the application. One example of user-interface tracking is seen in the work of Trafton and Brock [TB96] whose system provides a layer between the user interface and the application to keep track of the

user's actions, comparing them to an internal representation of various task models, to try to identify the task being performed by the user. When a correspondence can be pinned down, the user is offered the option of the sequence being completed automatically. Masson and De Keyser implemented a prototype of their *Cognitive Execution Support System* which anticipates errors and warns users when these errors could occur [MK93]. Yoshimune and Ogawa [YO94] developed a graphical feedback system which watches user interactions with a guide book, and suggests correct procedures if the user deviates from what is deemed to be an optimum procedure.

Fawcett and Provost [FP90] worked on finding ways to predict whether the user of a given account is not the authorised user. They profile each user by characterising behaviour based on histories of previous sessions. Myka *et al.* [MGS92] developed a system which automatically generates hypertexts and then records user actions when interacting with the text to determine whether any relationships can be inferred about the document by tracing user actions. Many researchers have studied the processes and patterns of user interaction with computer systems [BF88, CE89, LM88, Mar89, WSA97], while Lin *et al.* [LLM91] have developed methods for visualising the masses of data collected about user search patterns in a variety of graphical formats, allowing human pattern recognition capabilities to be applied.

5.4.2 Second Perspective — System-Level Monitoring

Other researchers have looked at tracking application use of system resources. Burton and Kelly [BK98, BK99] have developed a tool which traces system calls and provides the ability to re-execute these calls to allow system tuning.

Jeffery *et al.* [JZTB98] introduce the Alamo monitor program execution monitoring architecture which assists developers in bug-detection, profiling programs and visualisations. Siegle and Hofmann [SH92] have developed the SUPRENUM microprocessor which uses a hybrid combination of software and hardware monitoring to determine parallel program behaviour. This assists programmers in gaining insight into the execution of their parallel programs. Wybranietz and Haban [WH88] also use a hybrid approach to observe system behaviour, measure performance, and record system information. They make use of a special measurement processor which runs monitoring software for each distributed node in the system. The information thus derived is displayed graphically and used to improve understanding about run-time system behaviour. Joyce *et al.* [JLSU87] monitor distributed systems by means of a distributed programming environment called Jade, which assists the programmer in debugging, testing and evaluating distributed systems.

Eisenhauer and Schwan [ES98] have addressed the problems experienced as a result of the traditional event-stream mechanism that most monitoring devices use to report on activity. They propose that the communication, instead of only proceeding in one direction from the application to the monitoring program, should be flowing in both directions. They argue that the monitoring program should be able to send "steering" information back to

the application. This is facilitated by the use of augmented objects which will both send monitoring reports, and receive steering information.

When a decision is made to track an application, there are basically two ways of going about it — invasively and non-invasively. The following sections will discuss these alternative approaches.

5.4.3 First Approach — Invasive Tracking

If we consider an executing application, we can see that there are various levels at which tracking agents can be inserted into the system:

1. *Within the application itself* — This is probably the most common method of tracking application activity, as is demonstrated by Thomas [Tho96], and Welland *et al.* [WSA97], for example.

   Application-invasive tracking requires that a reporting component be inserted into the application code. This code is inserted either at development time or once the need for monitoring becomes evident. Ball and Larus [BL94] have described algorithms for placing code within programs in order to record program behaviour and performance.

   Inserting monitoring code could have negative effects. Errors can easily be introduced into the system by the monitoring code and it could be very difficult to locate these errors. More rarely, the insertion of monitoring code could actually remove errors from the system. This could be caused, for instance, by the fact that the monitoring code slows down the threads and problems which could occur when threads co-ordinate are alleviated.

   In order to disable the monitoring, the programmer must either remove the code or use some sort of environment variable or flag setting to disable the reporting. Either way, if the monitoring code is not removed, it will negatively affect the performance of the application. If it is removed, it is entirely possible that human fallibility will lead to more errors being made and cause much valuable time to be wasted in order to correct the error thus introduced into the system.

2. *Inside the libraries or classes the application uses* — Inserting tracking code into libraries will track the activity of the contents of that particular set of classes, not the application. Since other applications could use the same libraries, it is necessary either to duplicate the library and insert the reporting code into it, or disable the reporting when it is used by other applications. Thus in this case you would get library monitoring rather than application monitoring.

3. *Via the operating system* — This is even more generalised than library monitoring. Operating system monitoring will generate many reports about all and sundry events.

The monitoring application will have quite a job filtering out the meaningful reports from the dross.

All the techniques mentioned in this section are invasive in one way or another — and one can readily understand why this is so. There is a need to be invasive to get the amount of information required by the developers of systems, in order to perform the types of functions for which the tuning is required.

5.4.4 Second Approach — Non-invasive Tracking

The monitoring in this case should not make changes either to the source code, or make use of a non-standard set of libraries. Some examples are:

1. Using *reflection*, which must necessarily be language dependent, for example —

   (a) *Java:* Welch and Stroud [WS99] give a comprehensive overview of the various approaches to reflection in Java and note that most of them require access to source code, or the use of a customised Java Virtual Machine — the portable operating system used by Java programs. This does not meet the criterion of non-invasion, but the Kava approach described by Welch and Stroud does exercise reflection non-invasively. They use runtime byte code transformation in order to incorporate the use of special meta-object protocols, which are used for implementing special behaviour into the system. This mechanism could be used just as easily for application monitoring.

   (b) *Oberon-2:* Möessenböck and Steindl [MS99] describe a reflection technique for the Oberon-2 language which allows a programmer to access run-time information about variables and procedures, and allows the programmer to manipulate the values of such variables.

2. Using *proxies* — Chalmers *et al.* [CRB98] have developed a non-invasive methodology to build up Web usage histories for users in a particular community. The user search path is compared to paths of other users within the community and if a match is found, sites visited by the other users will be suggested as being of probable interest. Wexelblat and Maes [WM99] have built a set of tools to support Web browsing. These tools accumulate a history of other user's search paths and make it available to later users. Their tools contextualise the web pages the user is viewing.

3. Using *operating system APIs* — Some operating systems, such as Windows, have substantial APIs to support non-invasive tracking. An example of this is the DeskWatch facility.

4. Using *specialised hardware* — Argade *et al.* [ACT94] present a non-invasive technique for monitoring applications, but they use a specially tailored piece of hardware to

facilitate the monitoring. Their main goal is to simulate application execution, so that the application execution environment can be optimised.

5.4.5 Summary

To summarise, tracking can be carried out either invasively or non-invasively. Invasive tracking is risky, since it could introduce errors and be expensive in terms of time and effort to disable the reporting mechanism when there is no longer a need for it. It is also, by definition, application-specific, and tracking must be added to each application type individually. Non-invasive tracking is easily deactivated and can seamlessly track a variety of applications, but is much harder to accomplish.

Whereas the results of user interface monitoring are sometimes utilised by the end-user of a system [CRB98, YO94, MK93, TB96], it is often carried out primarily for the benefit of system developers and maintenance teams. System resource monitoring is carried out exclusively for the benefit of system development teams. One important stake-holder in application use, the end-user, is seldom catered for. This research will consider the provision of feedback for the benefit of the end-user to be a special concern— separated from the basic functionality concern of the application. This will be done by using the results of non-invasive application monitoring, implemented by means of proxies, to augment application feedback.

5.5 Third Mechanism — The Visualisation

Portraying information about application activity in order to augment application feedback is a novel use of the information derived from application tracking. The last important issue to be addressed concerns the manner in which the information thus obtained can be visualised in a helpful manner.

Chapter 4 argued for the provision of both immediate and archival feedback. Section 4.6 justified the need for the visualisation of application activity to provide the required feedback, rather than supplying merely textual feedback. The following subsections will discuss the research carried out in the visualisation area.

5.5.1 Visualisation of User Interaction with an Application

It has been noted by various researchers that discourse typically has an incremental quality about it [CM93, LM94]. This need is often satisfied in tutorial or visualisation applications by supplying the user with a log file containing previous explanations [EL96, DJA93]. This does not link the explanations to user actions, though, and is therefore of limited assistance in visualising the user interaction with the system.

There are three types of research which have some bearing here — the first is research into the visualisation of software execution; the second is research into the visualisation of

user-machine dialogue, and the third is the visualisation of serial information.

5.5.2 Visualising Execution of Software

Some researchers have worked on visualising the execution of software [ESS92, BDPS94, Jer96, KMS+95]. This is done primarily for the benefit of developers who need to analyse access patterns, and increase understanding of the program execution. Drew and Hendley [DH95] have worked on visualising complex object oriented software systems.

5.5.3 Visualising Dialogue

Other researchers have worked on visualisation techniques which maintain and present a record of user dialogue with the machine. This information can be used for providing a record of explanations, as shown by Lemaire and Moore in [LM94]. Kurlander *et al.* [KF90] illustrate a system which allows users to browse, redo or undo past actions which were performed using a graphical editor. Reiser *et al.* [RFG+88] developed a system which provides a record, in graphical format, of a student's solution to a problem.

Rich and Sidner [RS97] have developed a collaborative interface agent which maintains the history and context of the interaction between the user and the application. The agent interacts directly with the application, and with the user. It then maintains a history based on interaction with the user and observation of user actions. This agent queries the application, and makes recommendations based on observation of the user's interaction with the application. Two windows are used to facilitate the visualisation of the user's communication with the agent, and the agent's communication with the application. The communication is textual, based on an artificial language developed by Sidner [Sid94].

Berlage and Genau [BG93] developed the GINA framework, which provides a history mechanism for multi-user applications. This framework allows users who are located at different sites to work in collaboration on the same document. The framework requires the application programmer to provide additional hooks to facilitate the functioning of the framework.

5.5.4 Visualising serial periodic data

The data to be visualised — application interaction with the user — can be modeled as event-anchored serial periodic data [CK98]. This type of data has periods with different durations. Each period is composed of some user activity followed by some application activity caused by information supplied by the user activity. Each period is triggered by some user actions, signaled by events. The time taken for each period varies according to many factors. Periods follow each other in serial form, mirroring the serial nature of human processing capabilities.

Some researchers have worked on visualising different types of purely serial data. Some, such as Chi *et al.* [hCKBR97] have used tabular techniques. Rao and Card [RC94] and

Spenke *et al.* [SBB96] allow the user to interactively explore the data being displayed in a tabular format. Other researchers have worked on techniques for displaying serial data. One approach, by Robertson *et al.* [RM93], shows a "perspective wall", with time moving from left to right, and the central part of the wall giving the current focus. Plaisant *et al.* [PMR+96] developed LifeLines which shows a person's history compactly, with selectable items allowing the user to get more detail as required.

5.5.5 Interacting with the Visualisation

Serial data exploration is often supported by one of two tools — *dynamic querying* and *focus+context* techniques [CK98]. Dynamic query systems allow users to explore the data by executing queries, using user-friendly interfaces [KPS95]. What Carlis and Konstan call *focus+context* is the same as Shneiderman's [Shn98] *overview and zoom* approach. This approach displays a broad overview and allows the user to zoom in on items of interest. Some examples of research using this technique can be found in [Fur86, RM93, SSTR93].

Carlis and Konstan [CK98] present a scheme for visualising serial periodic data which displays data along a spiral so that both serial and periodic qualities of the data can become visible. They have also incorporated some dynamic querying facilities into their visualisation, feeling that it was not obvious how the focus and context technique would be applied.

5.5.6 Conclusion

Once again, the pitch of the research to be found in this area mirrors the approach taken in the application tracking field i.e. half of the work done benefits application developers — usually giving insights about the execution of the application program. The other work is done for the benefit of the end-user and depicts the user's interaction with the system in the form of a structure — usually a list – containing a representation of user commands. Examples of this are the selective paste offered by `Emacs`, or the `history` command used in Unix and MS-Dos. The author is unable to locate research which maps user dialogue to application response, independently of the application, to provide a visualisation for the benefit of the end-user.

5.6 Consolidation

The approach proposed here is use of application tracking to enable the programmer to treat feedback provision as a separate concern and to provide feedback by means of a visualisation of session activity. This approach has positive and negative points and it is as well to enumerate them here, before continuing with the design of the generic framework.

The proposed approach is made possible by the architecture, and generic features, of component-based systems. Specific details about the features exploited by this technique

will be covered in detail in the following chapter. The following sections address the benefits and limitations of the approach.

5.6.1 Benefits of the Proposed Approach

There are two basic techniques which make up the foundation for this approach: separation of concerns and application tracking. The benefits of using this combination are:

- The programmer's job is simplified.
  - Separating the feedback concern from the basic concern of the application reduces complexity and allows programmers to concentrate on the main task of the program — the functionality of the program [Kic96].
  - The programmer does not have to provide detailed feedback about application external errors throughout the application.
  - The programmer will not be required to anticipate all possible problems which could occur as a result of the failure semantics of the distributed system.
  - The programmer can get debugging assistance during application development. This is primarily linked to their use of the middle-tier components, since this use is observable, and can thus be reported.
  - The programmers need no longer be human-computer interaction experts, since much of the work will be done by the generic framework.
- The non-invasive approach requires minimal effort from the application programmer, which makes it more likely to be used.
- The feedback is augmented by means of non-invasive application tracking, which means that the application has the flexibility to function either with or without it, and the end-user can use it only when required, and deactivate it once the need disappears.
- Distributed systems open up the possibility of many more indeterminate failures, and it is therefore useful to have a standard way of indicating that an error has occurred, and for finding out more about that error [Str93].
- The generic framework will supply a feedback display which can act as an external memory aid to the user. This is what Norman [Nor98] calls "knowledge in the world", which makes it easier for the user to pick up the threads after an interruption or error.
- The feedback display can be designed to be extensible, which will make it easier to accommodate changing user needs.

5.6.2 Limitations of the Proposed Approach

The disadvantages of the approach are that:

- It can only give feedback based on the external activities of the application. Thus the feedback that can be provided is limited to the interaction of the application with the user and the rest of the distributed system.
- It requires the use of a language with introspective capabilities, since this is essential for the generation of proxies — the mechanism used to implement the non-invasive application tracking.
- It is bound to have a negative impact on the performance of the application. This matter is addressed fully in Chapter 8.

5.6.3 Summary

The problem definition rests on the central assumption that feedback provision is difficult and that it is seldom provided adequately and appropriately. The proposed solution is based on three supporting areas of research — separation of concerns, application tracking and visualisation — *and* the particular features of component-based systems, as is illustrated in Figure 5.1.

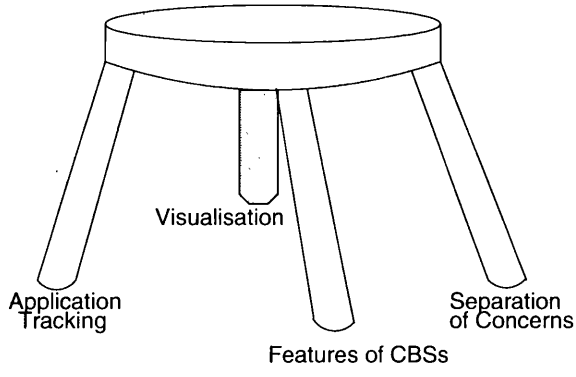

Figure 5.1: Supporting research

Each research area plays an equally important role without which the proposed solution would falter. Having decided to augment feedback by means of separating the concern, and tracking application activity in order to obtain enough information to provide that concern, it is necessary to test this by implementing a prototype of the framework. The implemented framework obtains information about application activity and provides a visualisation of that activity in order to augment the feedback provided by the application

The next part of this dissertation will describe the design and implementation of a prototype of the generic framework, which was used to test the proposal made in this Chapter.

part IV

HERCULE — Design and Implementation

I never worry about action, only inaction.

Sir Winston Churchill

If A is a success in life, then A equals x plus y plus z.
Work is x; y is play; and z is keeping your mouth shut.

Albert Einstein

Sometimes I think the surest sign that intelligent life exists elsewhere in the universe is that none of it has tried to contact us.

Bill Watterson

Calvin and Hobbes

chapter 6

HERCULE's Design

Part II of this dissertation described the problems experienced in providing adequate feedback in component-based systems. The proposed solution entails the use of a generic feedback-enhancing framework which works by tracking application activity and providing a visualisation of that activity in order to augment the feedback provided by the application. It has been argued that this framework would allow feedback to be treated as a separate concern, freeing the programmer to concentrate on the functionality of the application program.

The approach discussed here which has been applied to meet user feedback needs is applicable to a wide range of computer application systems. This research has focused on a feedback mechanism for component-based systems since these systems are distributed, increasing the likelihood of errors. The nature of component-based systems also decreases the likelihood of adequate feedback provision — as motivated in Chapter 5.

The concept of a framework was explored in Section 2.2.2, which concluded that a framework should provide a generic solution for a set of similar problems. The framework described in this chapter seeks to provide a generic solution to the problem of providing feedback in CBSs. The framework has been named HERCULE after Hercule Poirot, Agatha Christie's legendary detective — since it essentially acts as a detective which watches all events, tries to discover the reasons for quirks, and explain application activity.

This chapter will explore the rationale behind the design of HERCULE (Section 6.1). Sections 6.2 and 6.3 discuss the technology supporting HERCULE's observation and explanatory roles. The general architecture and functionality of the framework is described in Section 6.4 and the visualisation of the application activity is discussed in Section 6.5. Section 6.6 concludes the chapter. Chapter 7 will then go on to discuss the actual implementation details.

6.1 Design Philosophy

The purpose of this research is to provide a framework which facilitates the provision of a visualisation of the user's interaction with the application. The effect of this visualisation is to provide feedback including dynamic immediate feedback about the current state of the system, and archival feedback about previous states of the application.

6.1.1 Design Principles

A number of design decisions were made, each of which is described below.

- *Flexibility:* To allow *any* existing or new application to make use of a stand-alone generic feedback enhancing framework. The framework should not be tailored to a specific group of applications, except that broad category of thin-client systems which routinely appears in three-tier CBSs. The thin-client basically provides the *Graphical User Interface* (GUI) for the application, while the actual business processing is done by the other two tiers. This is not a particularly restrictive requirement, since most systems are moving to three-tiers in these days of e-commerce. By allowing existing applications to make use of the framework and thus obtaining the benefits of the extra feedback, it is hoped that the idea will become more widely accepted and that this will speed the uptake of the concept.

- *Painlessness:* To require minimal participation from the application programmer. This requirement is important because any extra burden on an application developer is unlikely to be appreciated and, even if the programmer is willing explicitly to invoke calls to HERCULE, this could be done incorrectly, which would result in the application becoming even less usable than the original version. Additionally, if the framework requires application programmer participation, existing applications would be disqualified from utilising its functionality.

  It is as well to be absolutely clear about the meaning of the word minimal, since it is a relative term. The approach which is intended here is that programmers would be able to rely on the framework to provide the extra feedback, but would have to take no action *within their programs* to facilitate it. They are also not to be expected

to participate in the insertion of any mechanisms into the system to facilitate the functioning of the framework that they would not have provided in any case.

What *is* expected is that they will participate in the tailoring of descriptive messages which are supplied to the end-user to describe what the system is doing. HERCULE can only give meaningful messages if assisted to do so by a human agent — and programmers are the most important and valuable allies in this respect, since they will become completely familiar with the server components' idiosyncrasies as they develop their program. This is the full extent of their participation.

- *Optionality:* If the user decides not to use the framework, it should not intrude on the system. This could be interpreted in two different ways:
  - The user could choose to have HERCULE running in the background, but make no use of the facility. The impact here is a slight performance penalty only.
  - On the other hand, the user could choose not to use it at all and simply execute the application without additional feedback. In this case the environmental variables must be altered, so that HERCULE would not activate at all. It would simply take up a little room on the hard disk, which is not a scarce resource.
- *Least damage:* The failure of HERCULE should not in any way cause the failure of the application. The negative impact of HERCULE on the application performance should also be kept to a minimum. It would be unreasonable to expect HERCULE to have no impact at all, since extra computation is being carried out by HERCULE. An endeavour was made to design HERCULE to have the smallest negative impact possible.
- *Non-invasiveness:* No part of the application should be changed to accommodate the framework. The alternative to this is that an application could be engineered to enable HERCULE, but that would invalidate optionality. Thus optionality and non-invasiveness go hand-in-hand — you cannot have one without the other.
- *Non-intrusiveness:* The HERCULE console should always be available, perhaps in the form of an icon, or on the screen in the form of a window, but, because of the aforementioned points, should not intrude. It must maintain an up-to-date display depicting information about session activities so that it can be used by the user as a feedback mechanism at any time. By "not intruding" what is meant is that the HERCULE display will not display itself, unbidden, in front of the application's windows, will not force its help on the user and will not make the user take any extraneous action to deactivate it.

  This is in stark contrast to the deplorable tendency of certain products to force help on the user in the form of the annoying paper clip. While one empathises with the

designer's probable good intentions in creating this facility, the expert user is often so alienated and aggravated by this unwanted assistance that it is more damaging than helpful[1].

- *Simplicity:* Complex schemes are admirable, but offer far more opportunities for disaster. Complexity leads to distracted effort, while simplicity leads to a more focused effort [dB98]. Designing with simplicity as the aim produces a more elegant, understandable solution, enabling the remaining time to be spent more profitably on mechanisms for visualising application activity.

- *Clarity:* Explanations should be understandable and lucid. This is no simple matter. We have all been the recipients of unintelligible messages — no matter how computer literate we are. The programmer can be of assistance in tailoring these messages, but that is not likely to be the ultimate solution. HERCULE should enable the post-implementation tailoring of explanations and messages so that one user's problem can be solved and then the explanation relayed to HERCULE on other machines so that the problem is solved for other users too.

- *Versatility:* It is often better to simplify a process than to train people to cope with complexity [dB98]. Explanations and error messages should be relayed at the user's level. Tailoring facilities should be provided which will offer different types of explanations and error messages dependent on the requirements of the user. If the user is an end-user with no interest in the inner functioning of the system, the explanations should be at a high level and, if the user is the system designer, the explanations should give far more detail.

6.1.2 Accessing HERCULE

A decision must be made about the facilitating mechanism used to provide the user with access to the feedback. It could be achieved in various ways:

- *activated by a special control sequence* from the keyboard. This might be daunting to technophobes or novice users, and might prevent the user from making use of it.

- *a button added on to the application's window* — perhaps at the bottom of the window — which allows the user to summon help. This conflicts with the non-invasiveness and optionality aims.

[1] It could be argued that expert users should know how to deactivate the paper clip. They do indeed, but when the help is offered they are engaged in another activity. Switching off the feature entails an interruption, together with the accompanying loss of context. Once the primary task has been completed the user will probably have forgotten about the paper clip until its next appearance.

- *in a minimised window*, which can be maximised as required. This would satisfy the design aims, but the window, being hidden, would not be in a position to offer dynamic feedback with respect to the state of the system.
- *in a window being displayed to the right of the user's screen.* This option was ultimately chosen since it facilitates the provision of both dynamic immediate and archival feedback at a glance. The fact that the user does not have to go and look for the feedback makes it immediately available and since it is always in the same place the user knows exactly where to find it.

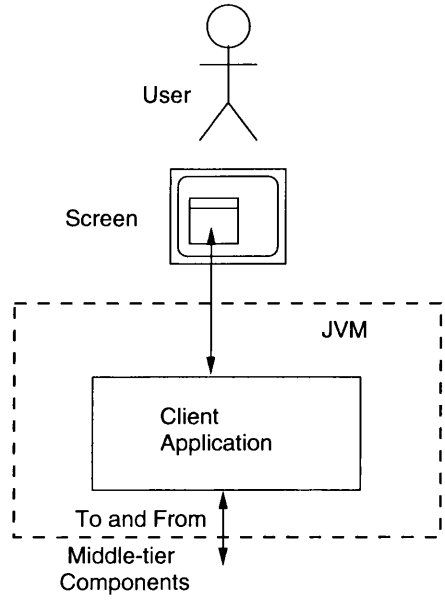

Figure 6.1: Application executing without HERCULE

The application running without HERCULE is shown in Figure 6.1, while when the application runs in harness with HERCULE, the structure of activity is shown in Figure 6.2.

6.1.3 Required Application Features

Before proceeding further, it is necessary to state exactly what is required, both of the application system and the programmer, to use HERCULE. The generic framework scheme relies on, and exploits, the following features of component-based systems:

1. Their tiered structure, with most of the processing being done in a different address space. The client application makes extensive use of "external" entities to carry out processing on its behalf.
2. The object-oriented nature of inter-tier communication. The client program issues requests to the middle tier and receives replies indicating the success or failure of the

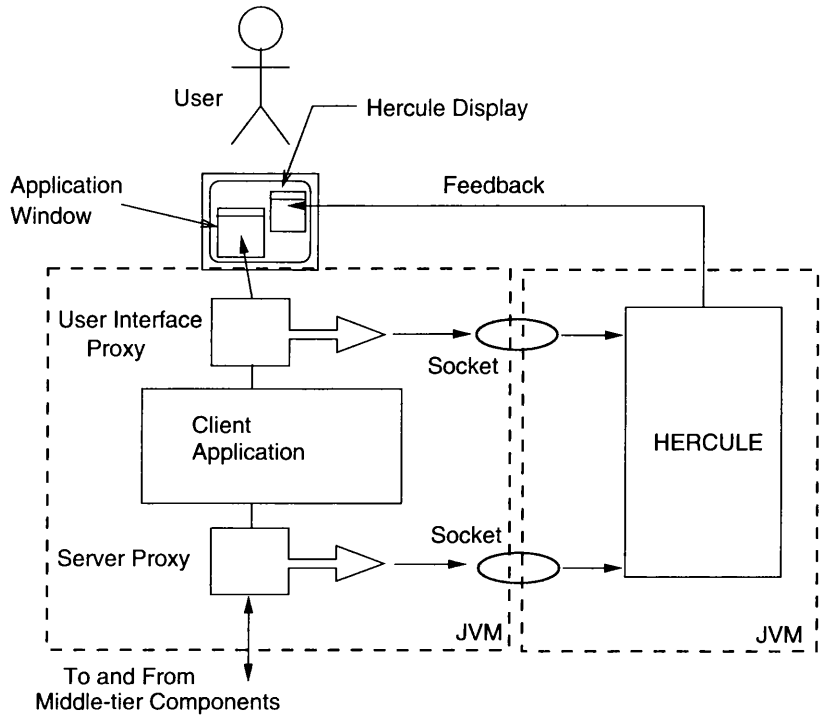

Figure 6.2: Application running in harness with HERCULE

processing carried out as a consequence.

3. The business logic provided by the middle tier of three-tier systems is often supplied by server components housed within an application server. This means that the middle-tier components, being independently developed, are: accessed via defined interfaces; which must be self-describing; and are accompanied by at least some form of documentation which can be harnessed by the framework. It also implies the existence of some sort of component documentation intended to inform the programmer of the functionality of the component.

4. The event-based nature of graphical user interfaces. It is therefore relatively simple to detect meaningful activity, from the application's point of view, at the user interface.

These features are essential in supporting an independent feedback facility since application behaviour must be observed and explanations supplied by means of a visualisation of activity based on interpretation of this observation. The first feature ensures that much of the application activity will be observable. The second ensures that the communication with the middle tier is easily understood, since it is structured in a predictable format. The third ensures that the essence of the communication thus observed can be interpreted correctly, while the fourth feature supplies the framework with an understanding of the relevance of events at the user interface.

The last requirement is that the application programmer must have the necessary expertise to be able to use HERCULE effectively to provide extra feedback. This means that the application programmer must have the required expertise both in Java and in EJBs. This is not an exacting requirement, since the programmer has to have this knowledge to build an end-user application for a component-based system anyway.

6.1.4 And Thus...

HERCULE needs two distinct facilitating functions: observation of the user and application activity; and explanation of that activity. The following section discusses HERCULE's observation function, while Section 6.3 explains how the components are described in order to give HERCULE information about method semantics, in order to fulfill its role of exponent.

6.2 Facilitating HERCULE's Observation Function

The aim of flexibility is satisfied by not making changes to either application code or any of the packages being used by the application. This ensures that any application can function either with or without HERCULE and also satisfies optionality and non-invasiveness.

Chapter 5 introduced the HERCULE concept, which is based on the observation of the external behaviour of an application. No attempt is made to deduce the internal functioning of the application. Thus HERCULE *observes* the application's interaction with the user, and with the rest of the CBS and merely reports on what it observes. The HERCULE approach thus monitors systems on an *application level* — specifically Java applications — rather than at a system level. This has been decided on for several reasons:

1. The application-oriented approach makes it possible to involve the programmer in tailoring messages for the end-user, because the semantics of the communication with the environment is easily understandable, as they are merely method invocations.

2. Application tracking using Java offers a platform-independent opportunity for monitoring, rather than system tracking, which is platform dependent. Platform independence is extremely important for thin-client distributed systems, since most CBSs are structured this way. The thin-client, especially the e-commerce thin-client, must be designed to be executed on any computer that could possibly connect to the middleware server. Most three-tier CBSs will have many different types of client accessing the middle tier, providing tailored clients for different needs. For example, the same middle tier could support a browser client, a Java application client and a telephone interface client.

   The CBS client application can therefore not really make any assumptions about the type of computer used as the platform for the client program. HERCULE is intended to be an end-user assistant and must therefore be able to run on any platform that

supports Java and uses it. System-level monitoring only works on a specific platform and is not the right option for HERCULE.

3. System-level monitoring is complex to achieve and it is very difficult to link events to the application activity. This difficulty is confirmed when you consider that all system-level monitoring done so far has delivered results to system engineers — not end-users. (See Section 5.4)

Since the application-oriented approach has been chosen, the mechanism to facilitate tracking needs to be decided. One way to track an application non-invasively is to insert proxies between the application and the environment — which requires no changes to be made to the application code. A proxy must be inserted between the user and the application user-interface and between the application and the middleware layer. The use of proxies, following the *decorator* or *proxy* design pattern [GHJV94], satisfies the aims of painlessness, optionality, simplicity and non-invasiveness. Least damage is guaranteed by ensuring that these proxies cannot cause the failure of the application. The implementation should be carried out in such a way that the proxies, upon encountering a problem, will simply revert to the "normal" behaviour of the system. They should no longer report anything and simply act as an empty channel through which the application communicates.

There is no question of the proxies, once activated, being removed at runtime since the application holds references to both proxies, without being aware of the fact. There is no way to update these references inside the application so the best approach is simply to cause the least possible damage and behave as a sleeper. Once the proxies become aware of an error condition they must immediately cease to report to HERCULE, so that the impact on performance is negligible. There are two types of proxies to be inserted:

- the user-interface proxy; and
- the component proxies.

The communication between the proxies and HERCULE can be made either synchronously or asynchronously. Synchronous communication is simply not viable in this case, since that would entail the application waiting for HERCULE to accept reports from the proxies and slow the application unnacceptably. Asynchronous communication using some asynchronous messaging system would have less impact on the application, but it is doubtful that dynamic immediate feedback can be guaranteed in this case. The "minimal impact proxy" design pattern, described in Section 6.2.1, was developed to provide a reusable solution to this problem.

Section 6.2.2 discusses the design of the user interface proxy, while Section 6.2.3 describes the proxies which intercept communication between the application and the middleware layer.

6.2.1 The "Minimal Impact Proxy" Design Pattern

This new design pattern was developed specially for the HERCULE framework. This pattern can be used to link proxies to receiving applications, at runtime, in order to track application activity and to facilitate reporting of activity with minimal impact on application performance

The insertion of proxies enables the observation of application activity without necessitating the alteration of application code. However, with monitoring becoming more common and the reasons for monitoring ever more justified, it is beneficial to identify a design pattern, namely the "Minimal Impact Proxy" pattern — a general solution to a problem in context [GHJV94] — to ensure that the proxy does not slow the application down too much. This section will identify the key aspects of this common design structure that make it useful for reuse.

Characteristics — This pattern has two distinct features, the first is the use of proxies between the application and some component making up its environment. This could be the user interface, a server, a database or whatever interaction needs to be monitored. The means for insertion of these proxies does not form part of this design pattern. The second feature, the feature with which this pattern is concerned, is the *linkage* of the proxies with an independent application which will receive the reports generated by the proxies and act upon them.

Intent — Linkage of inserted proxies to a monitoring application *with minimum performance degradation.*

Applicability — This pattern will be used when there is a need to track an application by making use solely of proxies.

Structure: Reports — Reports should be catered for by a single class type, with various subtypes for specialising reports. The specialisation could be used to reflect different types of activity or different types of objects or operations on objects. It is important to note that all fields in the report should be easily stored[2], so that it can be transmitted by means of the socket mechanism. This means that objects tracked from the application cannot necessarily be included in the report, unless the programmer is certain that such objects will not contain unserializable fields.

Structure: Linkage — Shown in Figure 6.3.

Participants —

- Proxies — observe the activity and generate reports.

[2] As Java is being used, serializing the report structure will be sufficient.

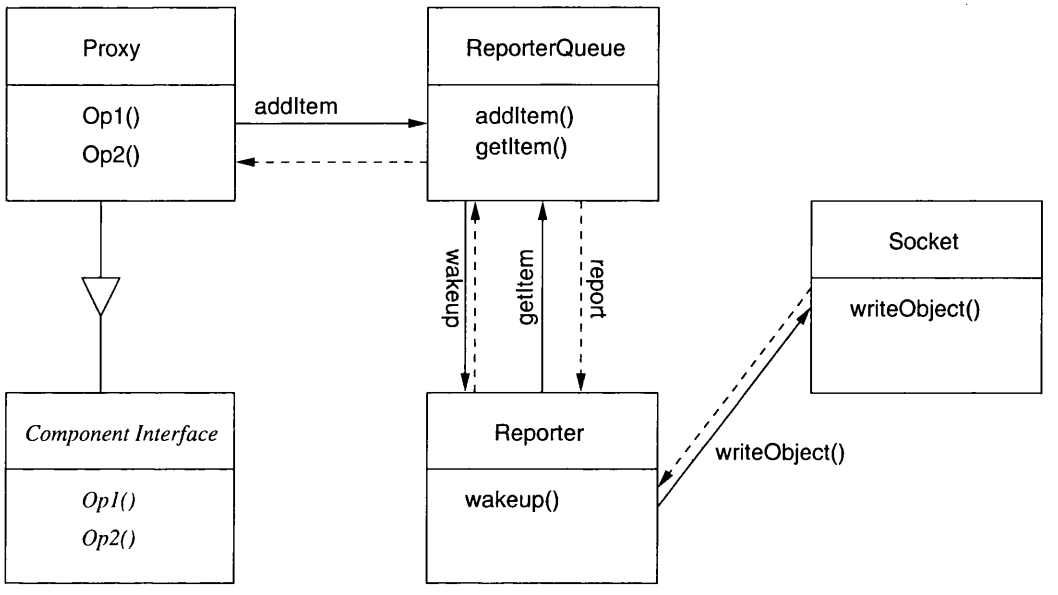

Figure 6.3: CBS Test Application Architecture

- `ReporterQueue` — provide a queueing structure, which introduces a measure of asynchronicity into the reporting activity — minimising the impact of the monitoring on the application.
- `Reporter` — removes the reports from the queue and sends the item to the monitoring application.
- Monitoring Application — which receives the reports and generates some meaningful representation with respect to the application activity.

Collaborations —

- The application unknowingly invokes methods on the proxies, who then report such activity and invoke methods on the actual components.
- The proxies put reports onto the `ReporterQueue` for forwarding to the monitoring application.
- The `ReporterQueue` notifies the `Reporter` of the existence of a report.
- The `Reporter` removes the report from the queue and sends it to the designated sockets.
- The monitoring application gets the reports from the designated socket.

Consequences — This pattern offers the following benefits:

1. If a non-invasive proxy insertion mechanism can be found, this is a great advantage. Even if some system libraries have to be altered to effect insertion of

proxies, the old libraries can be reinstated once monitoring is completed. There will be no problems with removing the monitoring code from the application.

2. The linkage structure ensures minimum impact on the application, since writing to the socket — which takes some time — is done asynchronously.

The following restrictions should be taken into account:

1. An insertion mechanism should be found which is not invasive. This is possible in Java, as will be shown in the following chapter, but it may not be as easy to achieve using other implementation languages.
2. This pattern uses two sockets, thereby tying them up. This might be a problem if any other application on the system uses the same socket numbers. This is an unavoidable consequence of the socket system and the user of the pattern should simply be aware of it, rather than waste time trying to overcome it.

Implementation — In implementing the linkage, the following should be noted:

1. In the interests of doing least damage, the proxies should not cease functioning if something goes wrong with the linkage. As can be seen from the `Reporter` Code Fragment in Appendix B, a global variable, `reportEvents`, is used which is initially set to `true`. If anything goes wrong with the connection, this variable is set to `false` and the entire object structure stays in place, acting as a channel through which messages are passed.
2. The `ReporterQueue` and `Reporter` run in their own threads, independently of the proxies, meaning that their failure will not cause failure of the application and that they can function without degrading the application's performance.
3. The environmental variable `verbose` is used to implement a measure of debugging in case the monitoring does not work. Thus, when a specialist is called in to ascertain the cause of a problem, the various error messages are easily generated without the need for a separate compilation. (This is applicable, once more, only to Java applications, since other languages have their own techniques for removing debugging-type output.)

Some of the implementation code is given in Appendix B.

6.2.2 The User-Interface Proxy

In order to track user activity, without being language specific, there are two requirements: the need to build up a data structure to represent the user interface; and the need to track activities by both the application and the user at that interface. These needs are addressed as follows:

1. The first is to build an *internal representation* of the user-interface structure. To achieve this, there is a need to know about each user-interface component being created and how the user interface is composed. In any user interface, the window is built up hierarchically. Each visible item on the screen is a *component*. Components have specialised functions. Some of these, the *container* components, have the ability to "house" other components. For example, in the window shown in Figure 6.4, the outer `Window` is a container component. It contains a *menu bar* (also a container) at the top containing four *menu* options. Each menu is also a container and holds the different menu item components. The window itself also contains three *panels*, the top one containing only a label, the second one containing four *button* components and the bottom one containing only the `Quit` button. A panel is a non-visible container which is used to group components together using some type of specific layout function.

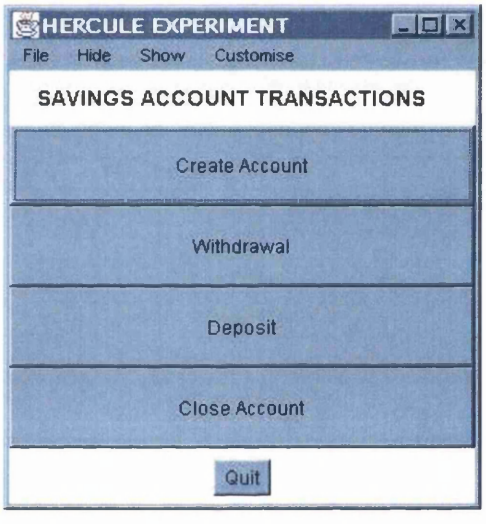

Figure 6.4: The Client User Interface

The Window in the figure houses the following components:

- a `MenuBar`, which in turn contains the following:
  - a File `Menu`
  - a Hide `Menu`
  - a Show `Menu`
  - a Customise `Menu`
- a `Panel` containing the `Label` "Savings Account Transactions"
- a `Panel` containing four `Buttons`:
  - Create Account
  - Withdrawal

- Deposit
- Close Account

- a `Panel` containing the Quit button

This can be represented as a tree structure, as shown in Figure 6.5.

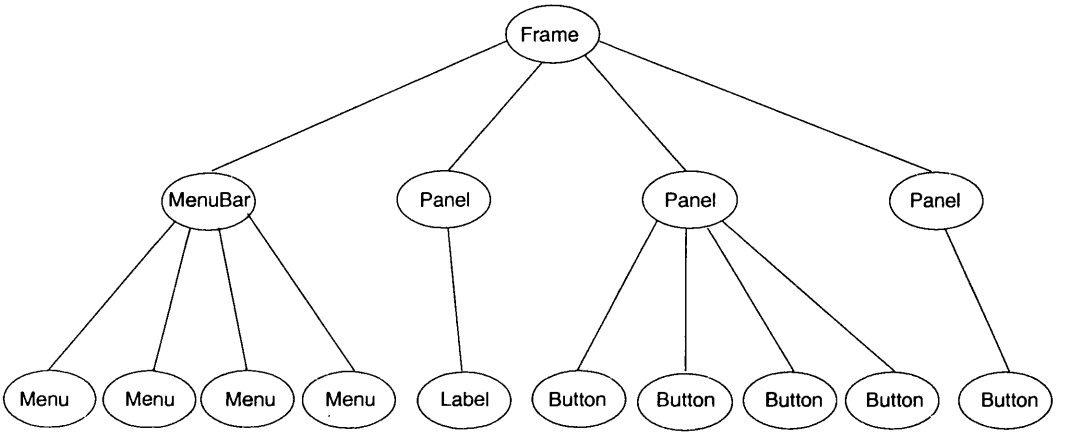

Figure 6.5: The Internal User Interface Representation

The user-interface tree structure is required so that the event delivery to contained components can be traced. Without such a structure, it would be more difficult to identify windows containing event-generating components, to keep track of components within a window being added or removed, and to provide any sort of context-sensitive feedback.

2. The second is to *keep track of activities* at the user interface — both with respect to the application and the user — and to associate them with the parts of the interface being used. To watch user and application activities, a tracking facility needs to be notified whenever the user does something at the GUI, and every time the application changes the appearance of the display. Since GUIs are primarily event-based, this information can reasonably be expected in the form of events. So, for example, in the `Window` shown in Figure 6.4, the application responds to button activations. In this case, HERCULE also needs to be apprised of button activations. On the other hand, in some cases, the application makes some on-screen components visible which were not visible before — or hides some components. This impacts on the user's view of the application's interface and is obviously important.

   HERCULE can only keep track of these activities if it is informed when actions occur. It could, upon learning that a component has been created, declare an interest in all events on that component. This would mean that it would be interested in every button press, every mouse movement, every key press, window activation and deactivation and much more. This volume of reporting would slow the system unacceptably.

The second-best option is to have HERCULE register an interest in events which are important to the application. These events would presumably precipitate some action on the part of the application and are therefore meaningful activities from the point of view of the user when using that particular application.

This keeps HERCULE aware of events triggered by user actions, but not activities triggered by the application. To keep track of these changes, HERCULE needs to register an interest in the visibility of components which could possibly be removed from, or added to, the display. The state of on-screen components which could have changing values are also of interest and HERCULE needs to be informed of these activities too. Since part of HERCULE's task is to provide a mechanism for rebuilding context, it is essential that HERCULE knows about any change in components that are visible at the user interface. Changes to invisible components are not important, since they will not have any effect on user perception.

The previous discussion has focused on the activities required to report on interaction between the user and *one* window display. An application typically makes use of many window structures in order to communicate with the user. Thus, HERCULE needs to be able to distinguish between different displayed windows and be aware of the transition between them. In addition to registering an interest in events which interest the application, or components within a window, HERCULE also has to register an interest in *windows* being made visible or invisible as the application executes.

Since there is only one user interface, HERCULE only needs one user-interface proxy — and it would have to be an intelligent agent, with specially tailored behaviour for each different type of user interface component. The behaviour of the proxy can be summarised as follows:

- For each component:
  - send a report signaling that the component has been created;
  - send a report giving the identity of the container the component resides in;
  - if the component has state, send a report about the state of the object. For example, a button's label would be reported.
  - if the component can be the source of events, check whether the application has registered an interest in events on the component and, if so, register an interest in those events too.
- For each container component:
  - register an interest in events on the container. This is so that HERCULE is informed of new components being added to, or removed from, the container. In some containers, the layout specifies that some components can be visible

while others can be invisible. Registering an interest in the container ensures that HERCULE is informed of components being added, removed or having their visibility altered.

- For each window component:
  - register an interest in all events on this window. This issues a report whenever a window is either shown, or closed, or destroyed.
  - register an interest in the simple activation and deactivation of a window. This is important because the user may switch to another application, to carry out some other work and then switch back to the application being tracked. HERCULE needs to know that this activation has not actually changed anything in the user interface and that it is merely a resumption of use after an interval.
  - report on the title of the window.

The reports generated by this scheme serve to keep HERCULE informed of the application and user activity at the interface, as well as events generated by the user. So, for example in the window in Figure 6.4, reports will be generated for each component — buttons, labels, panels, frames, menu bars and menus. The construction of the window is also reported, as for example, the fact that the menu bar is contained within the window frame. The state of the buttons and menus is reported too. The proxy registers an interest in each button and menu, since these are of interest to the application. An interest is also registered in the window itself, so that HERCULE is informed of window open and close activities. If the user clicks on the "Close Account" button, a report is sent to HERCULE informing it of the fact that the user had activated that button. If a new window was displayed, construction and status reports would be generated for this new window and a report generated to inform HERCULE that the first window was no longer active.

6.2.3 The Component Proxies

To intercept communication with server components, it is necessary to intercept each of three different phases of this communication:

1. when the client application "makes contact" with the application server;
2. when the actual component is being located; and
3. when methods are being invoked on the server components.

The first contact typically involves an object from a naming facility, while the second uses that object to locate server components. A naming facility is used in order to locate the required server component. This is done in different ways according to the component model being used, but each scheme has the basic use of a naming facility in common. If

HERCULE intercepts communication with the naming object by means of the insertion of a proxy into the system, the naming object can engineer the insertion of all the necessary server component proxies from there onwards since it is solely used to gain access to server components used by the application.

There is a need for a server component proxy for *each* interface of *each* server component. To satisfy the painlessness aim, proxies must be generated automatically. The programmer should not have to put any effort into getting the proxies written or installed into the system. To facilitate the creation and insertion of proxies, a language with extensive introspective qualities is required, for example, the Java reflection package [Mic99], which allows the investigation of all aspects of a component interface and enables generation of a proxy for any component implementing that interface.

The general structure of the proxy is essentially that of a "wrapper" [GHJV94]. The proxy implements the same interface as the component, so that the proxy instance can be substituted for the component instance. The application program uses the proxy as it would the component instance returned by the middleware server. The fact that the proxy implements the same interface as the component, and is compatible at a type level, makes this substitution possible. The interface inheritance mechanism makes it possible to substitute one object for a completely different object, as long as both implement the same interface or a subtype of the interface.

The proxy reports on all method invocations, then invokes the method on the actual component and reports on the value returned or exception thrown. The Java code for the proxy incorporating this functionality is generated automatically and then compiled so that the class files are automatically made available to the JVM at runtime. The proxies can be generated either at runtime or offline — Chapter 7 discusses this issue further. The proxies have the following functionality:

- When the proxy is initialised, a report is sent to HERCULE telling it about this server component interface and informing HERCULE of the name of the descriptor object for this interface.
- For the interface method signatures:
  1. store the parameters provided by the application program in a data structure;
  2. report that the method is about to be invoked;
  3. invoke the method on the wrapped component;
  4. report the completion of the method invocation — including the return values or the exception thrown.
- Ensure that *all* exceptions are caught, so that a report can be relayed to the proxy about it, before relaying it back to the application.

Once the proxies are in place, HERCULE needs to have some understanding of the semantics of method invocations, so that explanations can be generated for method invocations. The following section discusses the approach to this problem.

6.3 Facilitating HERCULE's Explanatory Function

In accordance with the aim of clarity, it would not be sufficient to report on methods invoked in the same format as, for example, an exception output statement, since that would not make any sense to the end-user. HERCULE, as an observer, has no understanding of the semantics of either the inputs supplied by the user, the methods invoked on the server components or the results from the method invocations. HERCULE therefore needs to have access to textual descriptions of these events, so that these descriptions can be relayed to the user as part of the feedback.

To get descriptive information about method invocations, existing component documentation is mined. HERCULE should function with the minimum requirements. HERCULE's requirements should not be greater than that which can be expected from a component supplier. Since there is presently no standard for documents supplied with components, the absolutely minimum requirements, without which no component would be delivered, are the following:

1. An *Application Programmer Interface* (API) document, explaining the purpose of the component, and giving details of method functionality, for example, `javadoc` [Mic98b] output.

2. One or more interface classes through which the component can be accessed.

3. A deployment document which specifies the context dependencies of the server component and explains how the component should be deployed.

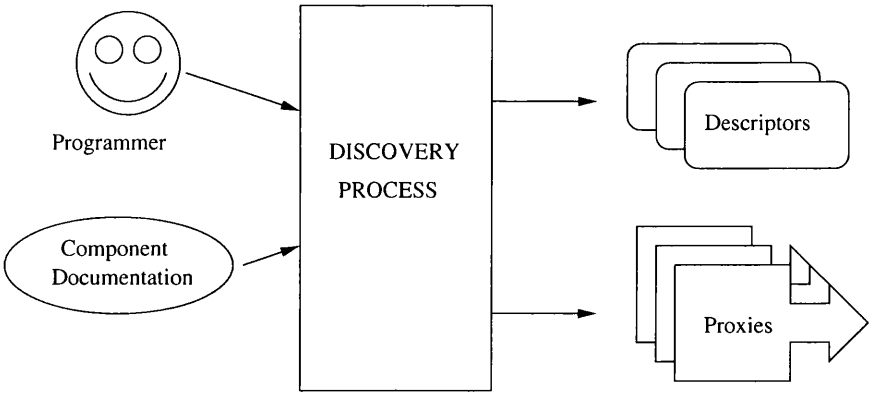

Figure 6.6: The HERCULE's discovery process

HERCULE mines the information from this documentation to customise itself with respect to that specific component. The explanations might not be suitable for an end-user and thus the programmer should be provided with a tool to allow these explanations to be augmented easily. To provide HERCULE with the required semantic information, a *discovery* process is executed to build up a set of descriptors for each participating server component interface. (see Figure 6.6) This descriptor holds information about:

1. the interface name;
2. the method signature for each method in the interface;
3. the semantics of each method invocation. This is a free-text description explaining what the method does; and
4. the possible errors and exceptions which could be produced by each invocation and an explanation for each particular error.

It is to be hoped that, in time, more descriptive component documentation will be delivered, as a matter of course with server components. The need for rich component specifications is critical [ND99] It is the component specification that allows component consumers to determine quickly which services are provided by the component [Sho98].

6.4 HERCULE's Architecture

The HERCULE framework essentially obtains details of the dialogue between the user and the application, together with an understanding of the effects of user actions — method invocations triggered by these actions. The framework must transform information about the dialogue to a graphical feedback display. The design of HERCULE was driven by the need to find the simplest and most elegant solution to the problem. Some complexity could not be avoided, as becomes evident from the discussion of the window manager component, but the structure of HERCULE, shown in Figure 6.7, was deliberately kept as unelaborate as possible, in accordance with the aim of simplicity. Each of the constituent components is explained in the following subsections.

6.4.1 Communication modules

The "Get UI Reports" & "Get Proxy Reports" modules receive user interface or proxy reports and make them available to the GoBetween module. Since two types of reports can be expected, there are two of these modules dedicated to receiving each individual report type — each on a different socket. To receive the reports, the following steps are taken:

- listen on the designated socket and wait for the application's proxy to make contact;

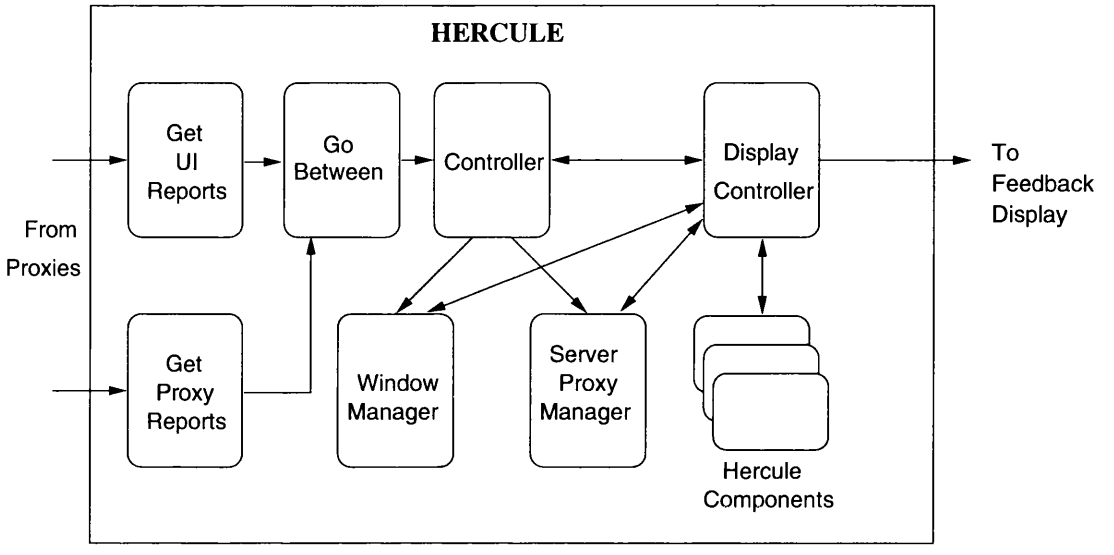

Figure 6.7: HERCULE's Internal Architecture

- maintain a queue of messages, appoint an object to wait at the socket for new messages and append the messages to the queue when they arrive;
- reply to requests from the GoBetween for messages by removing and returning the message from the head of the queue — if there is a message. If there are no current messages, simply wait a while and try again; and
- if communications break down, throw an exception to the controller, so that the display can be updated to reflect the fact that the application has severed the connection. This is probably an indication that the application has completed its execution.

This procedure is shown in Figure 6.8.

6.4.2 Controller

This is the control centre for HERCULE — the "brain" that controls and co-ordinates all activities. It is responsible for initialising all the other components at launch time and assigning each to a separate thread. HERCULE needs to be multi-threaded because the communication modules have to block while waiting for messages from the proxies and the HERCULE display must be able to respond in spite of this. Apart from this, the controller also launches the console and maintains the status display by informing it when the proxies make contact and when they sever contact at application completion time. During system operation, it requests reports from the communication modules and decides what to do with the reports. The controller relates the user interface reports to the proxy reports to link user interface activity to application reaction. If a user action at the user interface, as signaled by an event, directly precedes a call to the server, we can assign a *purpose* to a user action.

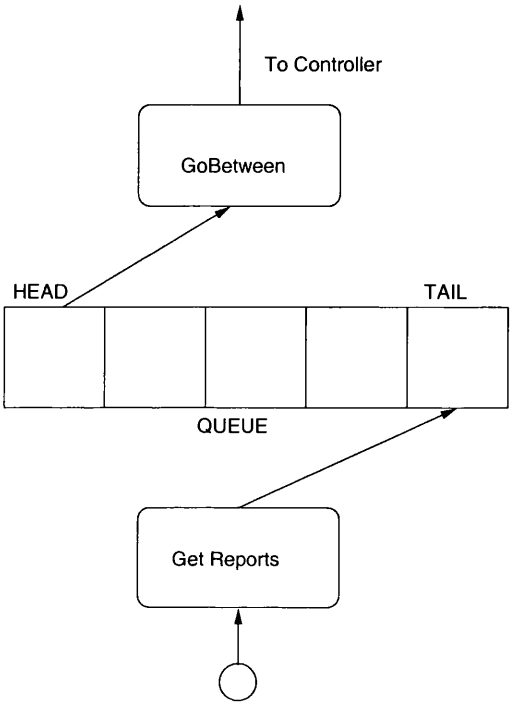

Figure 6.8: Communication Module

This must serve as a substitute for an understanding of the user's intention when the action was taken with respect to the sequence of proxy invocations thus triggered. Construction, status and event reports are sent to the Window Manager, while server proxy reports are sent to the Server Proxy Manager.

6.4.3 The Window Manager

The window manager has a dual function. Its first function is to build up an internal structure in memory to represent each individual window which is displayed — as indicated by construction reports. Linked to this are the status reports, which send details about the display characteristics of the windows, such as the text typed into a text field or the text displayed on a button. These reports, both construction and status, give a comprehensive picture of the appearance of the application user interface.

The second function is to keep track of user actions at the GUI. The window manager builds up a linked list of windows as they are created and displayed by the application and also remembers user actions with respect to those windows.

The window manager has a simple structure, shown in Figure 6.9, belying the complex nature of the software. In deciding on a mechanism for making sense of this plethora of information, the following decisions and assumptions were made:

1. It was assumed that the *structure* of a particular window would be constructed by

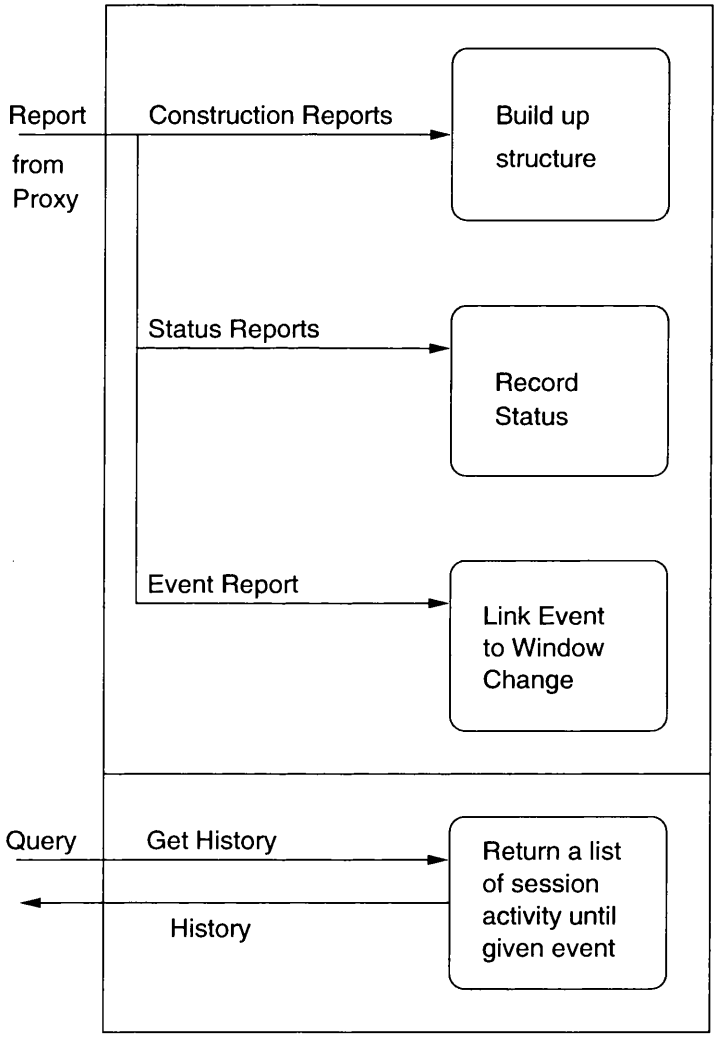

Figure 6.9: The Window Manager Architecture

the application only once, utilised as required and made invisible when no longer needed. Such a window could be re-displayed many times during the session activity, containing possibly different state in various user-interface components. For example, the application programmer could use the same window to issue warning messages or information messages. The same window could be used with different messages displayed in a particular message box. Thus, it is necessary to remember the structure of the different windows separately from the state. This window structure storage mechanism is illustrated in Figure 6.10.

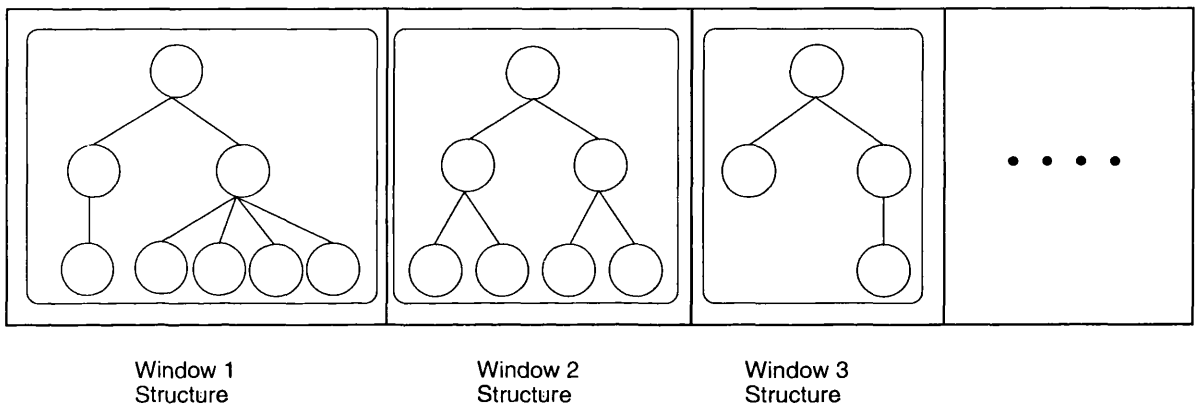

Figure 6.10: Storage of Window Structures

2. There are two ways of storing the window state (of visible components) separately from the structure of the window.

   - A duplicate structure could be stored with each node containing information about the state. For example, a node which represents a label could store the text value being displayed, while a container could store the layout structure which it uses to determine the layout of the composite components.
   - A list of components which have changed state could be stored.

   The choice between these two schemes would obviously be based on the type of application. An application with a very involved structure, but relatively few components which change state frequently, would benefit from the second approach. For example, a word processor window has many buttons and fixed menus, but only one main component which changes all the time — the editable text display.

   On the other hand, if the windows are re-used for various purposes with different entity states, or if a series of windows with different appearances are used to obtain information from the user, the first scheme is probably better.

   Since this software is generic, it is not easy to judge the nature of the applications so that a good choice can be made. However, this decision can once again be based on the

nature of thin-client systems. These systems generally collect information and then send the data to the middleware to be processed. This type of application generally uses a window structure of one type to get a specific type of information and then proceeds to another window display to get another type of information. This can be judged from the structure of systems like the online bookseller, `amazon.co.uk`, or any of the many e-stores in existence today which are a prime example of thin-client technology. Thus it was decided that the first scheme would be followed. The aforementioned scheme for storing state is illustrated in Figure 6.11.

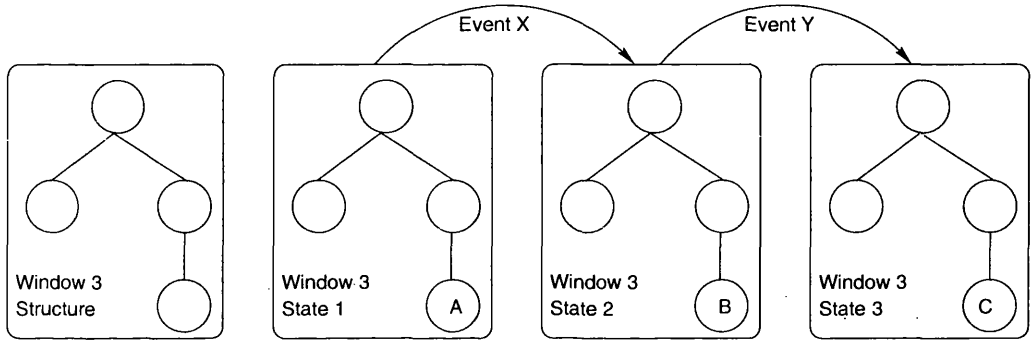

Figure 6.11: Storage of Window State Changes

3. Information about the state of the window components, constituting the window appearance, needs to be recorded. A decision must be made about the *extent* of information to be recorded. It is possible to store information about every possible feature — including colour, size, position on the screen, fonts used in the display and so on. To store this information so that it can be relayed to the user would obviously be valuable, but once again we come up against the fact that the negative impact of HERCULE on the application should be as small as possible. Each type of information stored leads to information being relayed between the proxies and HERCULE and slows the application down. Thus, a pragmatic approach was followed, with a minimum of information extracted and other details regretfully ignored. Therefore the state of each component with respect to displayed text and visibility on the screen is recorded, while the other features like colour, position and size are not collected.

Finally, the window manager has to satisfy a query relating to the session history. In storing the session activity history, it is not enough to store the succession of windows and window component state. An assumption about the event driven nature of the system has been made, thus it is reasonable to assume that some action by the user precipitates the transition either from one window type to another (shown proceeding from top to bottom in Figure 6.12), or from one state to another state in the same window (shown proceeding from left to right in the Figure 6.12). Therefore the state changes must be stored in conjunction

with the actions which caused them.

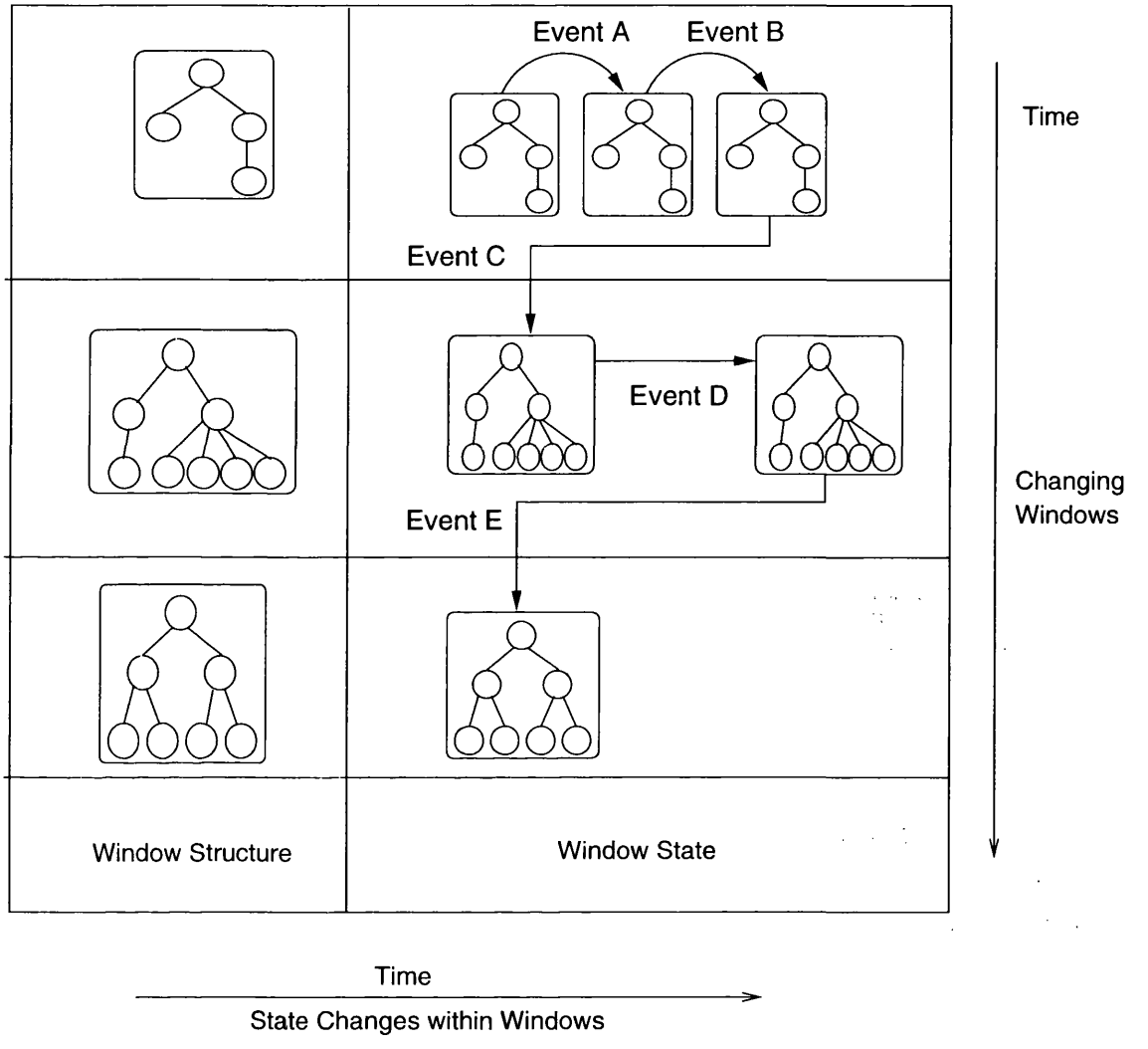

Figure 6.12: Storage of Session History

6.4.4 The Server Proxy Manager

This module keeps a history of all server component method invocations. There are three possible consequences of a call to the server:

1. no response within an expected time period;
2. an exception, signaling that an error occurred; or
3. correct execution, signaled by a return value or values to the user.

In the first case, the lack of a response within the expected time triggers investigation into the source of the delay. It is difficult to determine the difference between a slow server and a

dead server. This manager therefore keeps a record of reaction times. If the current reaction time exceeds double the current maximum time, it is assumed that the server has crashed. To diagnose the problem, the module firstly attempts to check whether the server is indeed still responding by attempting to establish a new connection to the server. If this fails, the manager then executes a program which checks whether the machine housing the server is functioning. The diagnosis will be reported to the user. It is readily acknowledged that diagnosis is not always possible, but it is hoped that experience with this framework will suggest better and more reliable ways of making a more conclusive and reliable diagnosis.

In the second and third cases the return values are stored together with the details of the call to augment the history of the session. In the second case, an exception handler is activated to investigate the source of the error. The descriptors will contain textual descriptions of the reasons for each exception thrown by method invocations.

The server proxy manager architecture is shown in Figure 6.13. The inputs received consist of reports generated by the proxies. The reports indicate one of four events:

- that a connection to a server has been made — giving the host and port details;
- that a specific component interface has been used by the application — giving information about the descriptor class which describes this component interface;
- that a method has been invoked on a component interface; or
- that a method invocation has completed, giving the return value or exception thrown.

In each case the information is stored for later availability. The third case causes a Timer to be started, which times the response and registers the absence of a response in the rare cases when this happens. In these cases the server proxy manager is informed so that information can be relayed to the display controller. The following queries are satisfied by the server proxy manager:

- getting an explanation for an exception thrown by a component;
- getting a list of method invocations which were precipitated by specified user actions.

The server proxy manager has another function too — that of maintaining a *system state indicator* which is an essential part of the immediate feedback to be provided by the HERCULE display. The server proxy manager is in a unique position to gauge the "health" of the rest of the distributed system. The system state indicator clearly shows whether the server is ready and waiting for work, busy servicing a request, or not responding. The server proxy manager is the first to know of any problems in this respect and therefore informs the display controller of the required state to be depicted.

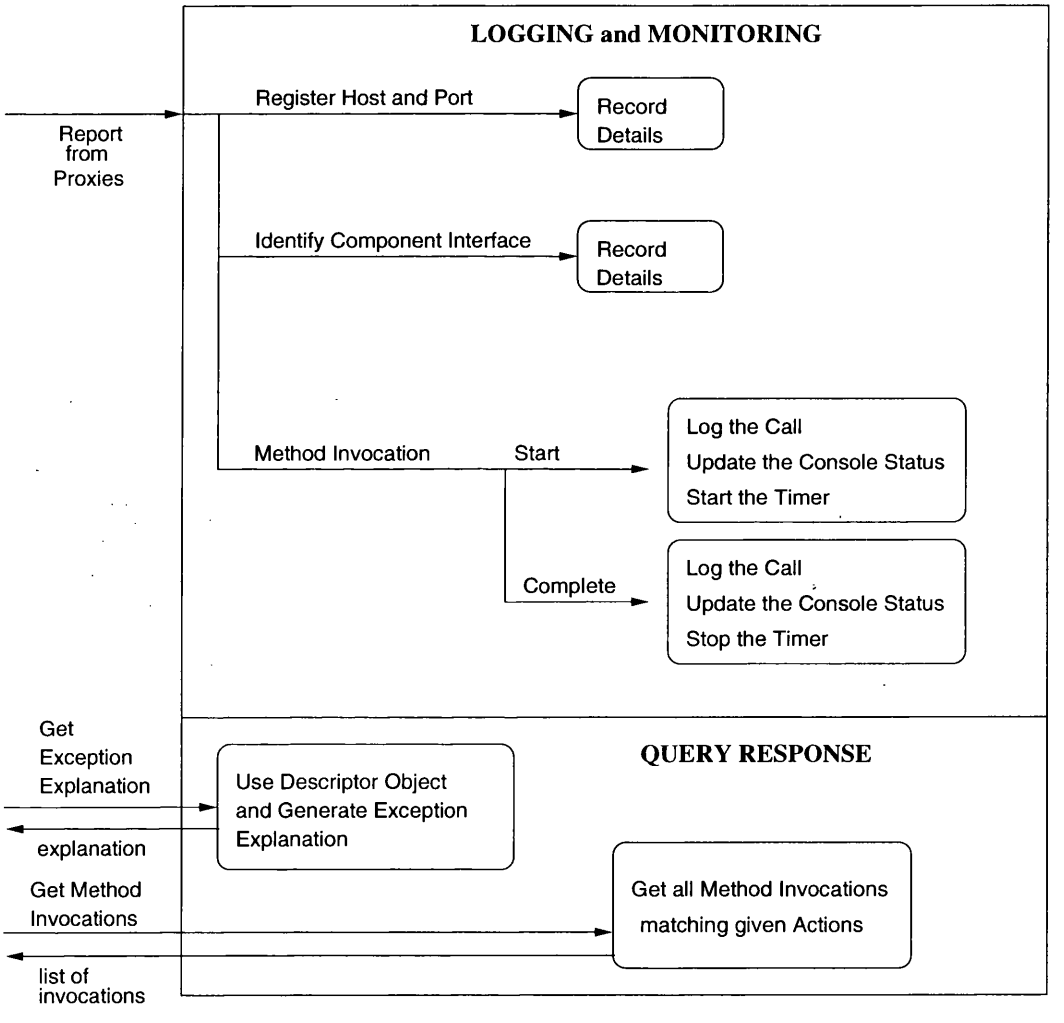

Figure 6.13: Architecture of the Server Proxy Manager

6.4.5 The Display Controller

This controller handles the display that provides both immediate and archival feedback. It depicts all activity for the session and can also explain errors and offer possible reasons for those errors. The display is active continuously, but does not intrude. If the user wants to verify any actions or get explanations of errors, the display can be consulted.

The session visualisation should depict the relationship between the user's actions and the actions of the system as a result. The system interactions with the rest of the CBS as a result of user actions occur in the form of global method invocations. The visualisation aspect is discussed in Section 6.5.

6.4.6 Hercule Components

These feedback-tailoring components extend the feedback capacity of the display and satisfy the versatility design principle. They can be tailored to the specific needs of the end-user, as discussed in Sections 4.5.3 and 4.5.4. There is a capacity to add them to the system dynamically so that the system can keep up with changing user needs.

Thus, suppose the system has been in use for some time and a blind user needs to make use of the system. A special feedback component which plays an audio message as explanation of system actions could be very helpful for such a user. This feedback component can be added dynamically to the HERCULE display on the new user's machine whereupon it would be available for use immediately. To support this extensibility of the HERCULE console, the following mechanism, shown in Figure 6.14, is used:

- An abstract class named `HerculeComponent` (which extends `java.awt.Panel`). This class must be implemented by any feedback component to be incorporated into the HERCULE console.

- A `HistoryListener` interface. The feedback component implements this interface and registers as a listener with the history panel. The feedback component is then notified of user actions at the history panel, which enables it to provide relevant feedback. The feedback component implements this interface if it is going to provide dynamic feedback related to a specific user activity.

- An `OutcomeListener` interface. The feedback component implements this interface and registers as a listener with the history panel. The feedback component is then notified of the outcome of system actions which were caused by a set of user actions. The feedback component typically implements this interface if it wants to provide statistics about the entire session activity, or performance.

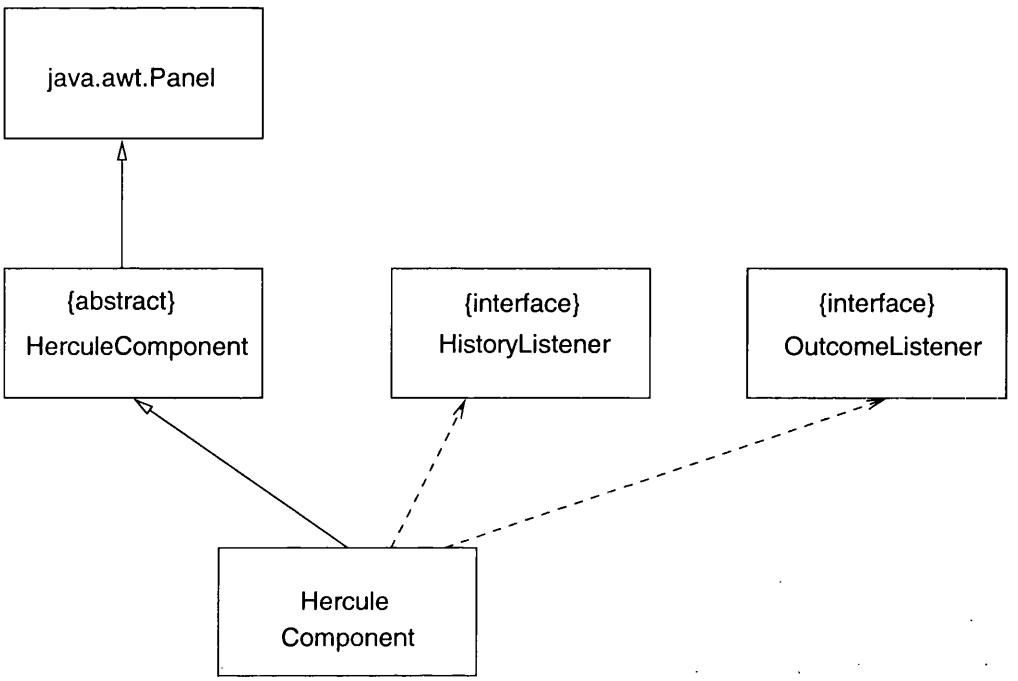

Figure 6.14: Structure for extending the HERCULE console

6.5 Application Activity Visualisation

The focus of this research has been end-users — consideration of their needs and attentiveness towards finding out how system interpretability can be enhanced. The purpose of this section is to explain the design of the application-activity visualisation in graphical format to satisfy these needs. The cost to the user of accessing this information is made up of the cost of finding it on the screen and the cost of assimilating it [CRM91]. To reduce the first cost, it should be available at a glance while to address the second, the information being depicted should not be ambiguous. The user should be left in no doubt of which particular action it refers to. This section therefore starts off by taking a look at the user's needs and summarising the findings of Chapter 4.

General Needs

Section 4.5.2 gave a summary of the user's feedback needs. There are some important things to be remembered in providing these:

- The feedback display should not intrude, but offer the user assistance. Thus, it should use as little screen space as possible. Many feedback devices tend to become overpowering and the last thing we want to do is to annoy.
- The user should be able to obtain as much information as possible immediately — and more if needed, but there is a need to be careful not to overload the user with

information.

- Allow different types of navigation of archival feedback.
- Section 4.4 concludes that feedback should be continuous. In Section 4.5, a distinction was made between immediate and archival feedback. Immediate feedback must necessarily be continuous and informative. Archival feedback should provide an immediate overview and summary of information — and then allow the user to interact with it in order to reconstruct inter-referential relationships between their input and the application's response (output).

Contextual Needs

It is often necessary to help the user reconstruct their context. This need was mentioned in the sections of Chapter 3 dealing with interruptions and errors. It was also referred to indirectly in the discussion of situated action in Chapter 4. The following analogies, with which we are all familiar, illustrate the need:

- when you go into a room to fetch something and, having arrived, you forget what it was you wanted. By going back to where you were, it is often possible to reconstruct the train of thought that prompted the errand.
- when you lose something you can try to reconstruct the events surrounding the last time you used the item. This often helps you to remember where the item is.

If the user is operating with an objective in mind [Suc87], rather than a rigid plan of action, and is responding to the system's state during the user of the application, the reconstruction of this state is extremely important in enabling the user to rebuild the circumstances that prompted action in the first place. A feedback mechanism can be truly helpful to the user in reconstructing mental context, by facilitating backtracking.

6.5.1 How Should the Application Activity Visualisation be Provided?

Section 4.6 argued the need for graphical rather than textual feedback. In providing such a visualisation it is necessary to build a model of the user's interaction with the application and to convert that to some sort of visualisation which is helpful and meaningful. Chen points out that there are two issues to be resolved [Che99], the structure of the information and its visualisation.

Structure of the Information

It is useful to examine the nature of the information to be depicted. HERCULE holds information about the appearance of the user interfaces, events at that interface and method invocations resulting from those events. This is a continuous process with one set of events

continuing on from method invocations resulting from previous events and so on. In choosing a structure to be visualised, it is important to find a configuration which can exploit the user's intuitive understanding of interaction with the application. The fact that HERCULE is working with a thin client greatly simplifies matters, since the application itself, by its very nature, only collects the necessary information and relays it to the middleware layer and does the minimum of processing itself. The user's operating paradigm is that some inputs will be supplied whereupon the application responds to those inputs by doing something meaningful. The pervasiveness of web browsers makes this paradigm well understood by many computer users and thus makes the task of designing this part of HERCULE somewhat simpler.

In broadly analysing the application activity, two types of activities stand out: the user-interface activity and the application communication with the middle tier. Chapter 4 compared the interaction of a user with the computer to a two-way conversation. In a conversation there are also two types of "activity" — what was said by one person and what was said by the other. In observing a conversation we can only guess at the internal reasoning process of the participants, based on what is said. In the same way, HERCULE, by tracking the application, only monitors external application behaviour and cannot attempt to guess at internal functioning of the application. Application interaction with the rest of the CBS occurs by means of method invocations.

Since not all user-interface activity results in server component method invocations, there could be a number of user-interface activities occurring before a method invocation. In the same way, a whole string of method invocations could be precipitated by a sequence of user-interface activities. The user carries out a set of actions and these actions change the state of the system in some way. These changes can be considered to be a *trace* of the user's actions. Suchman analyses the structure of discourse as follows: *"the user's actions can be grouped in a series of displays such that the last action prescribed by each display produces an effect that is detectable by the system, thereby initiating the process that produces the next display"* [Suc87].

To model this behaviour, it is necessary to consider each application thread in turn, with the sequence of user-interface activities (including user actions and application displays) which precede some or other component-based application-activity being called a *UI-sequence* (User Interface Sequence), and the series of method invocations thus precipitated being referred to as an *MI-sequence* (Method Invocation Sequence). When a UI-sequence is matched to an MI-sequence, we can call this mapping an *Episode.* This is illustrated in Fig. 6.15.

HERCULE must treat the dialogue as an information source that can be browsed by the user, thus giving a representation to the dialogue history. This visualisation is not merely a matter of displaying the content of user-interface activity, but needs to be linked to the method invocations which were precipitated as a result of the dialogue as well as the user interface activities. The system must have a strategy for producing tools which

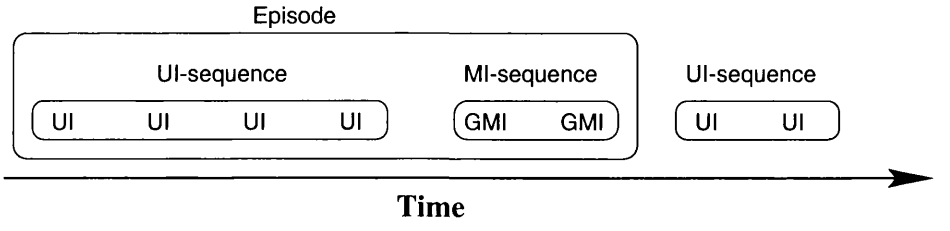

Figure 6.15: UI-sequences, MI-sequences and an Episode

allow the user to browse and search through the dialogue. Considering the conversational nature of the user's interaction with the application, it is possible to interpret the UI-sequences to be the application's interaction with the user, with the MI-sequences being the application's response to the user has instructed it to do. Thus HERCULE reveals the application's response to user inputs. In a conversation the listener is sometimes instructed to do something and the actions as a result of the instruction serve to inform the instructor of the understanding of the instructions given. Since the application's actions are often hidden and therefore unintelligible to the user, the user is left puzzled. By visualising this activity — making visible what is invisible — HERCULE can promote a better understanding of application functioning.

6.5.2 Visual Representation

Section 4.5.2 discussed the feedback features which should be provided by a computer application. These are satisfied as follows:

Immediate Feedback

This is satisfied by giving the user immediate feedback about the state of the system. Since the current time can be indicative of the state, that too is included here. The following displays are used to provide the required feedback:

- a status display;
- a current time display;
- explanations:
  - an explanation of the latest Episode (if chosen by the user); and
  - detailed information about latest method invocations (if chosen by the programmer). This requires nothing more from the programmer than a choice from a menu on HERCULE's display — whereupon the method invocations are displayed.

Archival Feedback

Archival feedback is best satisfied by an interface which allows the user to get details of the most recent interface activity, and the ability to access details of previous interaction. To satisfy archival feedback needs, therefore, the user must be given an overview of all session activities, with the option of getting an explanation of any one of the activities, as follows:

- access to facilitate reconstruction of context (memory aid)
- summary information (overview)
- an overview of Episodes (overview) — which can be expanded to:
  - time when Episode occurred;
  - explanation of an Episode;
  - detailed information about method invocations.
- an expanding facility (detail-on-demand) in which the system:
  - allows a choice of which Episode is to be expanded;
  - allows a choice of the type of expansion that is required — either end-user explanations or method invocation information for the programmer or both;
  - makes it easier for a programmer to add new feedback features to cater for different feedback requirements since new needs can be identified at any time.

6.5.3 Layout

The derived layout is shown in Figure 6.16. This layout has components to address each of the feedback needs outlined in the previous section as follows (small letters in brackets refer to the specified areas in the figure):

1. Immediate Feedback:
   - A status display (a). A *symbol* is used to depict the system status. This is used both to save space since a legend becomes unnecessary and to save the user time, since nothing needs to be read but the symbol can simply be interpreted directly.
   - A current time display (b), which is given in hh:mm am/pm format. The decision to display the time in digital rather than analog format was made for two reasons:
     (a) Many younger people today are not as familiar with analog watches as used to be the case and it is not as easy for them to tell the time at a glance. The digital display is suitable for all age groups.
     (b) There is a need for the user to compare the current time with the action time, displayed as part of the archival feedback. This is easier to do with digital displays.

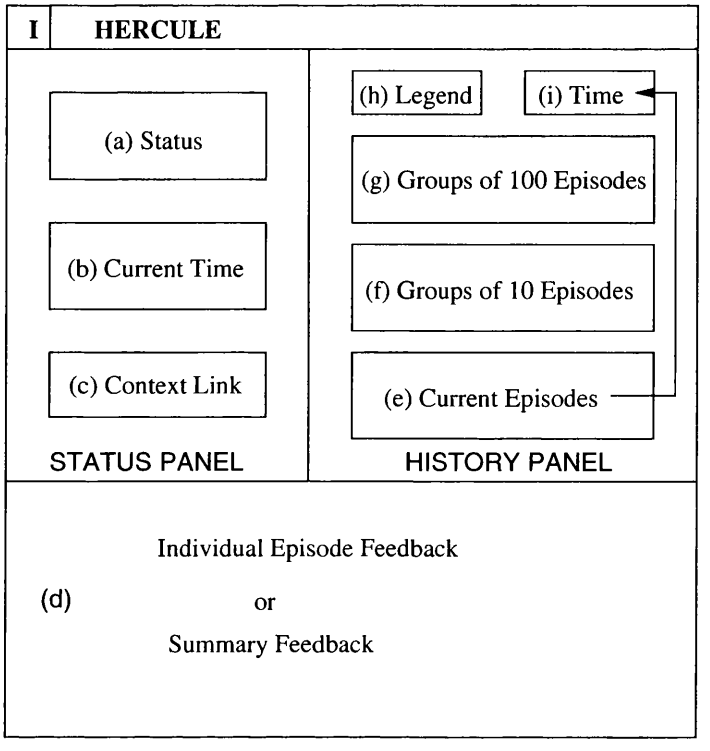

Figure 6.16: The Feedback Layout

- Explanations are supplied in the panel at the bottom of the display (d).

2. Archival Feedback:

   - Access to facilitate reconstruction of context (c) — supplied as a button. This reconstruction of context requires the animation of the window displays as they appeared on the screen. This is impossible and inadvisable to depict in a small space, so it is constructed on demand.
   - Summary information — supplied in the panel at the bottom of the display (d).
   - Overview of Episodes is supplied in the history panel as groups of episodes. They are not called episodes since the term "Episode" has only been coined to assist the designer in building a model of dialogue structure and using that term in the display would only confuse the user. Therefore they are referred to as *actions*, since the user is surely aware of their actions having an effect on the application.
   - Expanding Facility — obtained by clicking on one of the symbols used to depict episodes in the groups of episodes. This causes the explanation to be displayed in the panel at the bottom of the window. The topmost display (g) groups Episodes by hundreds, while the middle display, (f), groups them by tens. The bottom area (e) displays the current ten Episodes. Each episode has a link to:

- the time of the user action (i); and
- the explanation shown in the lower area (d).

The history panel contains grouped Episode areas which link downwards to the group presently being displayed.

This layout has the following desirable features, mentioned in Section 4.6.3 and as discussed by Vanderdonckt and Gillo [VG94]:

- *regularity*, as characterised by the fact that the components of the layout are determined by some evident principle.
- *vertical and horizontal alignment*, giving a pleasant aspect.
- *proportion* as shown by the various labeled areas, with no one area overwhelming the others.
- *horizontality*, since the display is wider than it is long. Vanderdonckt and Gillo cite research which shows that displays should have a greater width than height, as this is preferred by users.
- *simplicity and economy*, with only absolutely essential features being shown. Cluttered displays do nothing to ease understanding so that simplicity has been applied here as throughout the design phase.
- *unity*, with only one window being used to display all required information. The various components of the display are related to each other, giving an overall picture of application activity.
- *grouping*, which has been used to demonstrate an overview — areas for the Episodes are grouped together on the right. The status panel contains areas for status, current time and context linkage to provide immediate feedback with respect to the system state. Each panel provides a grouped area of feedback components, the status panel depicting immediate feedback with respect to the current status of the system and the history panel providing archival feedback.
- *sequentiality* and predictably arranged in a logical rhythmic order — by time. The feedback components will, by their similarity, help the user to anticipate their use and understand the relationship with the history panel.

6.5.4 Customisation

In Section 4.5.3 some differences between people were cited. It is tempting to satisfy these needs by providing a profuse collection of customisation facilities. There is evidence that people often take no advantage of customisation features, seeing the time spent on this

as time wasted [Mac91]. MacLean *et al.* [MCLM90] point out that users with extensive computer skills tended to make more use of customisation facilities than users who had no interest in the computer but just wanted to get on with their task. The latter users seem to have less expectation of tailoring their system. MacLean *et al.* advise that tailoring mechanisms be made more accessible to the user, to reduce the need to learn a new set of skills merely to customise the system.

In designing customisation features for the HERCULE display, the need for simplicity once again becomes paramount. There are two types of customisation to be considered — appearance and functionality. Customisation of appearance could possibly apply to the size of the fonts used or the colours used by the display. Customisation of functionality would have more to do with the actual feedback provided, like the type of explanations being tailored to the end-user or the programmer. The former should be accessible to the end-user, while the latter should be provided by the programmer and offered to the end-user as a possible option.

The initial prototype customisation features are kept to a minimum. It is certainly possible that experience with HERCULE will suggest the desirability of other customisation features and it would be interesting to investigate these needs at a later stage. The provisos mentioned in Section 4.5.3 are catered for as follows:

- Appearance:
  - *Physical abilities and physical workspaces.* HERCULE allows the user to choose whether an error should be signaled by a beep sound or not. This allows the user to adapt HERCULE to noisy environments and individual preferences with respect to beeps.
  - *Disabilities.* There is scope for HERCULE to offer an audible explanatory message instead of a textual one. This is handled by the addition of a new feedback component which is displayed at the bottom of the HERCULE display. The mechanism for doing this has been designed into HERCULE, but implementation of the actual audio feedback has been reserved for future attention.
  - *Elderly users.* The HERCULE display does not explicitly define font sizes and thus uses the default size defined by the user for the desktop. This means that HERCULE reflects the user's display preferences with respect to font and window size.
- Functionality:
  - *Cognitive and perceptual abilities.* Possible limitations are alleviated by the information given in the session history panel and by the use of a symbol to communicate system state.

- *Personality differences.* HERCULE seeks to make the application less threatening by explaining activities. It also seeks to reassure by offering a dynamic system-state indicator.
- *Cultural and international diversity.* The customisation feature offered by HERCULE allows the programmer to tailor messages to support these differences. Extra feedback components can also be developed specifically to support the user with unusual needs.

Design problems often appear to have many solutions. While solutions can often be compared to each other to find some which are better or worse than each other, it is often impossible to cite the best design while it might be uneconomical to expend a vast amount of time chasing after such an elusive design. Thus designers will often expend what they feel is a reasonable amount of effort, using guidelines such as the ones cited above, and arrive at a satisfactory design. Simon [Sim69] calls this "satisficing" — the process of seeking good or satisfactory solutions instead of optimal ones. The science of information visualisation is young enough to support this paradigm for the present, while the future may well produce stronger guidelines which allow us to approach the optimal solution more quickly.

6.6 Conclusion

The design of HERCULE suggests the need for three distinct tools:

1. A descriptor tool, which would:
   - provide a mechanism to generate descriptor objects which describe the server components used by an application. This should be generated automatically from the component documentation; and
   - provide a mechanism for explanatory messages to be updated during the lifetime of the system, by means of a simple interface.
2. A proxy generator, which would:
   - provide a mechanism for component interface proxies to be generated automatically.
3. A runtime feedback tool, which would:
   - intercept all server calls and keep a history of the calls to provide session feedback;
   - build up an internal representation of the user interface and watch all user activity at that interface;
   - provide runtime support for application users by providing continuous feedback and error explanations; and

- allow end-user and programmer customisation of the feedback display.

In addition, there is a need to devise a scheme for inserting the proxies dynamically so that the end-user does not have to bother about achieving this. The following chapter will discuss the details of the implementation of these tools and outline the mechanism used to insert the proxies.

Debugging is anticipated with distaste, performed with reluctance, and bragged about forever.
Anon.

Backup not found: (A)bort, (R)etry, (P)anic.
Anon.

chapter 7

Implementation

The design having been completed, the next step is to implement a prototype of HERCULE. Before details about implementation can be given, Section 7.1 describes the component-based test application framework within which HERCULE was implemented.

Section 7.2 will discuss the implementation of the user interface proxy, while Section 7.3 gives details about the technique for the automatic generation of server component proxies. The design chapter concluded that three tools were needed in order to facilitate HERCULE:

1. *A descriptor tool* — described in Section 7.4.1, providing a mechanism to automatically generate descriptor objects describing server components. It also provides a mechanism for updating explanatory messages by means of a simple interface.

2. *A proxy generator* — described in Section 7.4.2, providing a mechanism to generate component interface proxies automatically.

3. *A runtime feedback tool* — described in Section 7.5, builds up an internal representation of the user interface, tracks user interface activity, links it to requests for server activity and provides runtime support for application users by providing a visualisation of application activity.

Since the approach followed in this research has been to provide the feedback by means of a visualisation of application activity, an entire section, Section 7.6, has been devoted to this. Section 7.7 concludes the chapter.

7.1 Prototype Application

The prototype was tested on a three tier CBS, as shown in Figure 7.1, with *Enterprise Java Beans* (EJBs) [Mic98a] fulfilling the role of the server components. The application server used was the Tengah server from Weblogic [Tho98b], an all-Java application server. The test system was composed of a client on an NT host running on a Pentium 166, the Tengah server running on Solaris on a Pentium 166, with the third level being made up by a Cloudscape database [Wil99] containing a set of client accounts.

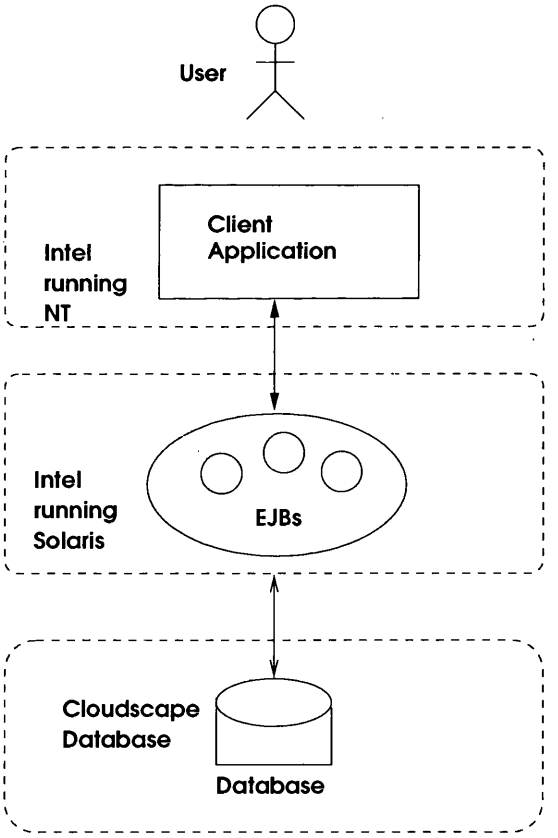

Figure 7.1: CBS Test Application Architecture

This system, which is typical of a three-tier CBS, was used to test the design of HERCULE. Although the test application is physically divided into three tiers with each tier running on a different machine, this is not necessary for the functioning of HERCULE. All three tiers could easily run on the same machine. All that is required to support HERCULE is that the client should be "thin" — meaning that most business-logic is taken care of by

another layer of the system.

The choice of EJBs to provide the middle tier was completely arbitrary with respect to functionality provided by the middle tier. All that was required was a middle tier to provide the business-logic layer. It could have been provided by either COM or CORBA components.

However, there were some other factors which led to the choice of EJBs. It was decided that a prototype based on COM objects would be too platform-specific. The delay in the CORBA Component Specification loaded the decision in favour of EJBs. Furthermore, the need for an implementation language with introspective capabilities, such as Java, made EJBs the obvious choice.

7.2 Observing User-Interface Activity

This section describes how to insert a user-interface proxy, positioned as shown in Figure 7.2. The first goal of the implementation is to intercept user activity successfully. This

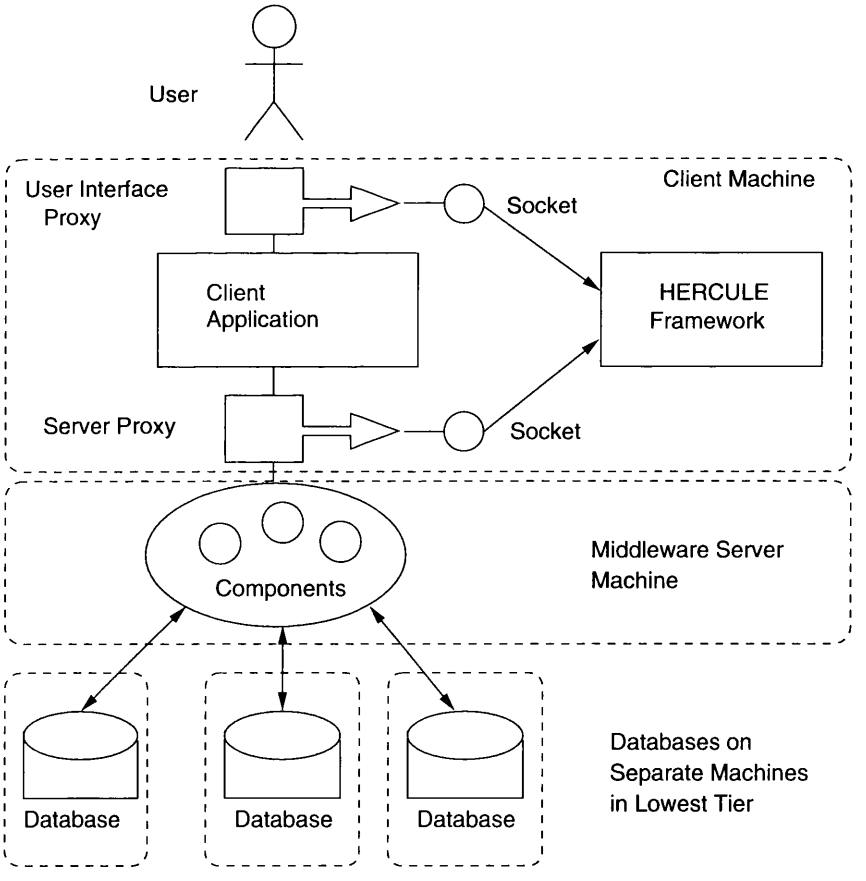

Figure 7.2: CBS Application Architecture with Proxies

involves two tasks: building a description of the active user interface and recording the user's

interaction with that interface. In a Java application, the interface consists of a hierarchy of interaction objects — instances of `awt` or `Swing` package classes — such as frames, panels and buttons. This hierarchy is built up by instantiating `awt` or `Swing` classes. User interaction results in calls to methods of these classes.

These tasks require HERCULE to be aware of the instantiation of new user interface components and to be informed when the state of any of these components changes. Fortunately, the Java runtime system enables the interception of component instantiation by means of the insertion of a special proxy object which is invoked when the user interface is being constructed. An adaptation of the minimal proxy impact pattern (Section 6.2.1) is used to allow the `ReporterQueue` object to register an interest in components of the user interface which are subject to change. This will be described for the case of intercepting button press events, but equivalent techniques apply to other user interface components too.

Section 7.2.1 describes the mechanism used by Java in providing platform-independent user-interface classes. Section 7.2.2 explains how the user-interface proxy is inserted into the system. Section 7.2.3 describes the operation of the proxy. Section 7.2.4 describes the mechanism used to watch and record user activity at the user interface, and Section 7.2.5 briefly describes how the reports about this activity are used.

7.2.1 Java Platform-Independent User-Interface Mechanism

The way that the JVM provides platform-independent user-interface classes for the GUI is by means of a combination of the `java.awt.Toolkit` class and a library of platform dependent `Toolkit` classes. When a Java program instantiates user-interface components in order to build a GUI, the component class instance will use the `Toolkit` to establish a link to a platform dependent peer.

When the Java application interacts with these `java.awt` objects, the messages are relayed to platform dependent peers, in order to display the required GUI. The peers handle all details so that the programmer is completely oblivious of the process. The programmer simply instantiates and invokes methods on the `java.awt` objects, while subsequent calls to the peer objects are completely invisible. The `java.awt.Toolkit` class has the responsibility for loading the platform dependent classes. This `Toolkit` is loaded automatically by the `java.awt` classes when they are instantiated. A programmer will often never have to make direct use of this class at all. For example, the program may include the following:

```
Button quit = new Button("Quit");
```

The `Button` class calls on the `Toolkit` to create the platform dependent peer object, `ButtonPeer`. This object is the actual platform specific object which is displayed on the user interface. If the programmer now calls:

```
quit.setLabel("Cancel");
```

then the `quit` object will call the `setLabel` method on `ButtonPeer` so that the label on the button on the GUI will change. The structure of this activity is shown in Figure 7.3.

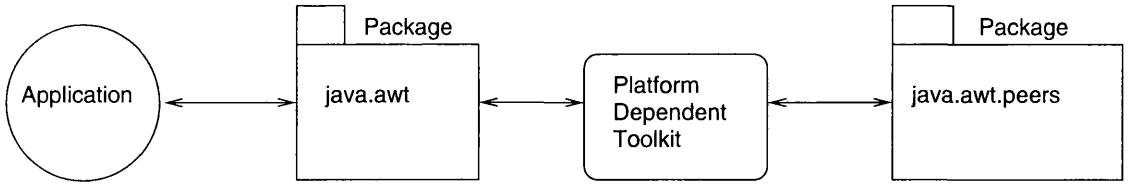

Figure 7.3: The Use of the Toolkit to facilitate GUI platform independence

7.2.2 Inserting the Proxy

The aim is to track user interface activity with respect to an application, without making changes to either the application, the `java` packages' source code or bytecode. A first approach would be to generate wrappers for all the classes in the `java.awt` package, use an auxiliary class loader and, by an additional level of indirection, substitute the proxy classes for the wrapped classes[1]. This satisfies the requirement that no part of the application should be altered and it also does not interfere with the `java.awt` package. Unfortunately we cannot wrap the `java.awt` package, because its use invokes the `java.awt.Toolkit` class. This class cannot be wrapped since it is abstract and therefore cannot be instantiated, so that the platform dependent `java.awt.Toolkit` and `java.awt` peers are loaded by the *application* class loader. This confuses the wrapped classes which are loaded by their own separate class loader, so that they consequently cannot reference the `Toolkit`. Since the `java.awt` package is essential for our purpose in tracking user interface activity[2], another mechanism must be used.

The approach just described attempted to intercept user interface communications for *each* user interface component. However, an alternative position for the interception of information can be found in the `Toolkit` class, since Java requires the creation of all user interface objects be created using this class. Two factors make this a viable proposal:

1. The first is that `Toolkit` is an abstract class. As an instance of an abstract class cannot be instantiated, the programmer either has to use an instance of a class that extends the abstract class or a static method which returns an instance of a subtype. The `java.awt` package makes use of the abstract class `java.awt.Toolkit`, which provides a static `getDefaultToolkit()` method. This gets the name of the platform dependent `Toolkit` class from system properties and obtains an instance of that class from the platform-specific libraries to be returned to the caller.

2. The second, which relies on the first, is that the static method `getDefaultToolkit()` allows the use of an environment variable (`-Dawt.toolkit=...`) to specify which `Toolkit` is to be loaded [Beg99]. The `java.awt.Toolkit` incorporates a mechanism

[1] This method is explained in detail in [RE00].

[2] `Swing` is built on top of the `awt` package, so applications using Swing also utilise the `awt` classes.

to allow the developer to substitute another `Toolkit` for the one which would, by default, be loaded by the JVM.

So, suppose a *proxy* `Toolkit` is written which extends `java.awt.Toolkit`, called `java.awt.ProxyToolkit`[3]. The `EssentialApp` application can be told to use this proxy `Toolkit` by starting the application with the following command line:

```
java -Dawt.toolkit=java.awt.ProxyToolkit EssentialApp
```

The `java.awt.ProxyToolkit` will be instantiated when the application needs an instance of a `Toolkit` and the proxy will thereby be dynamically activated.

7.2.3 The User-Interface Proxy

The `java.awt.ProxyToolkit` class, which extends `java.awt.Toolkit`, is the user interface proxy. When the application calls the static `getDefaultToolkit` method to get an instance of the toolkit, an instance of the `ProxyToolkit` is created. This `ProxyToolkit` then loads the OS specific `Toolkit`, so that the `ProxyToolkit` acts as a channel, relaying all calls to the platform dependent toolkit and relaying all return values back to the application. The resulting structure is shown in Figure 7.4.

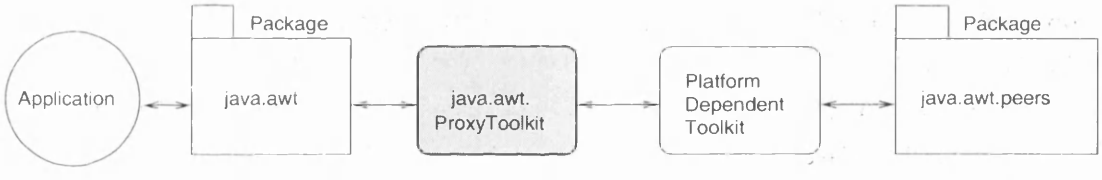

Figure 7.4: The System using the ProxyToolkit

Since the `ProxyToolkit` must *be* a `Toolkit` and re-route all method invocations on the `Toolkit`, it must implement *all* the public methods provided by the `java.awt.Toolkit` class. Within the `ProxyToolkit`, a static block loads the platform dependent toolkit, and maintains a reference to this toolkit so that all future method calls can be relayed to the platform dependent toolkit. The code is shown in Code Fragment 7.1.

All methods invoked on the system's default toolkit are forwarded to `ProxyToolkit`, which relays them to the platform dependent `Toolkit`. The `createButton` method in `ProxyToolkit` called when a `Button` is created, illustrates this.

The proxy can execute programmer-defined code in the overridden methods, which provides a means of extracting meaningful information from the `ProxyToolkit`, as required. It is important that application performance is not affected unduly by the presence of the `ProxyToolkit`. When reporting the required information the application should not be slowed down any more than is absolutely necessary. The minimal impact proxy pattern

[3]The `ProxyToolkit` *must* be part of the `java.awt` package because all the methods in the abstract `Toolkit` class are `protected`, and cannot be invoked by a member of another package — exactly what this proxy needs to do, in order to relay messages to the platform-dependent toolkit.

```
package java.awt;

public class ProxyToolkit extends Toolkit  {

    // the link to the platform dependent toolkit
    private static Toolkit theToolkit;
    // queue structure for reports
    ReporterQueue queue;

    // static block to initialise the "real" toolkit
    static {
      String toolkitName="";
      String osName = System.getProperty("os.name");

      // hardcode the names of the platform dependent toolkits here
      if (osName.indexOf("Windows")>=0) toolkitName = "sun.awt.windows.WToolkit";
      else if (osName.equals("Solaris")) toolkitName = "sun.awt.motif.MToolkit";

      try { theToolkit = (Toolkit)Class.forName(toolkitName).newInstance();}
      catch (Exception e) {
         e.printStackTrace();
         System.exit(0);
      } // catch
   } // static block

   protected ButtonPeer createButton(Button target) {
      queue.addItem(target,target.getParent());
      return theToolkit.createButton(target);
   } // createButton

  // rest of methods ....

} // ProxyToolkit
```

Code Fragment 7.1: ProxyToolkit

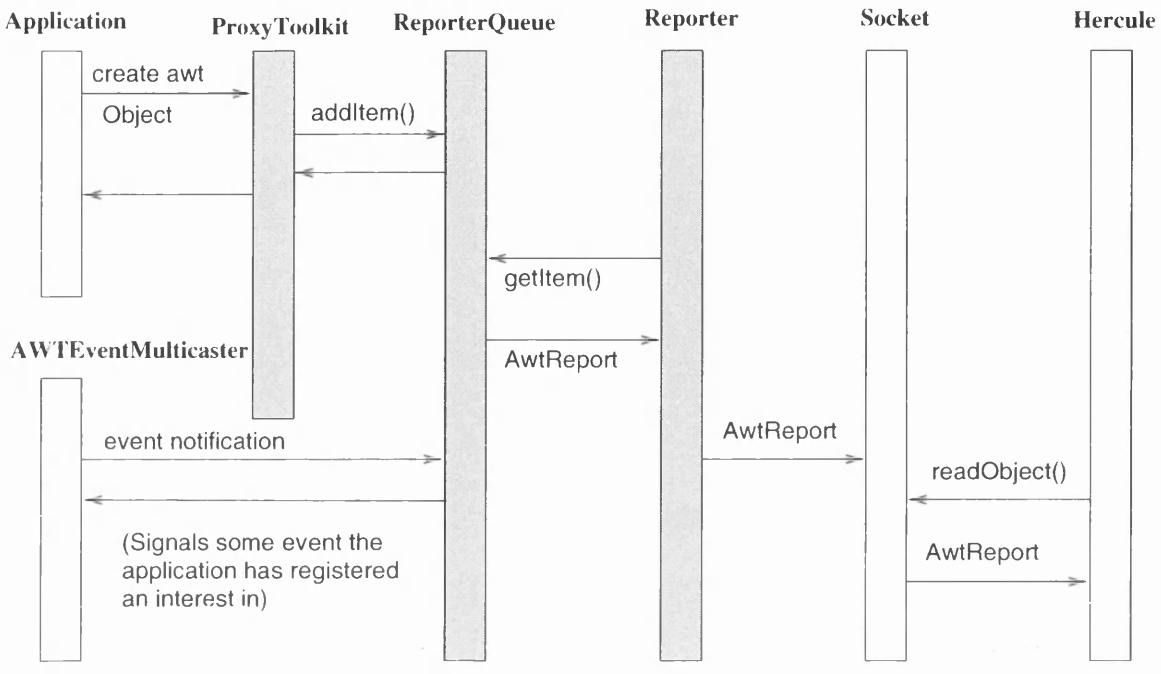

Figure 7.5: Structure of User Interface Reporting

described in Section 6.2.1 will be applied to utilise two distinct objects, the `ReporterQueue` and the `Reporter`, to ensure that the proxy has a minimal impact on the overall performance of the application.

The interaction between the `ProxyToolkit` and these two objects is shown in Figure 7.5. So, for example, if a `Button` is being created, and the `createButton` method is called in the `ProxyToolkit`, the `createButton` method would put an item on the queue describing the new item being created, as shown in the given code. So, for example, if a button, with the title `Quit`, is being created, two reports will be generated:

1. a *"new Component"* report to indicate that a button with the title `Quit`, has been created.

2. an *"add Component to Container"* report to indicate that the button resides in some specific panel container.

Information can now easily be extracted about the structure and composition of the user interface, enabling the construction of an internal structure duplicating each window structure. This structure provides the basis for making sense of user activity reports.

7.2.4 Watching User Activity

Once an internal structure has been created, the next requirement is to be able to keep track of user activities. This can only be done if HERCULE is informed when those actions occur.

HERCULE could, upon learning that a component has been created, declare an interest in all events upon that component. This would mean that HERCULE would be interested in every button press, every mouse movement, every key press, window activation and deactivation, and much more. This volume of reporting would slow the system unacceptably. The second best option is to register an interest in events which interest the application. These events would presumably precipitate some action on the part of the application and are therefore meaningful activities from the point of view of the user when using that particular application.

All `java.awt` components allow other objects to declare an interest in events on the component by registering as a *listener*. Each component has different capabilities so, for instance, a `java.awt.Button` has registered *action listeners* (registering, for example, the pressing of a button), while a `java.awt.TextComponent` has both *action listeners* and *text listeners*. The actions of interest are the pressing of the `Enter` key and the text listeners register all changes in the displayed text of the text component. The event notifications received as a result of registering as a listener will serve to provide a tangible record of all user activity.

When a component is instantiated via a call to the `Toolkit`, the `ProxyToolkit` will check whether the application has registered an interest in that component. If it has, the `ReporterQueue` will be added as a listener for that event. The `ReporterQueue` implements the interfaces for all listeners so that it has the ability to be registered as a listener for all types of user interface events. When the `ReporterQueue` receives an event notification from the `AWTEventMulticaster` (as shown in Figure 7.5), an *event* report will be placed on the queue, giving information about the type of event and the component that generated it. When this functionality has been included, the `createButton` method is altered as shown in Code Fragment 7.2.

```
protected ButtonPeer createButton(Button target) {
   // send a report through about this button,
   // as well as the button container
   queue.addItem(target,target.getParent());

   // is the application interested in this component as a source of
   // events? If so, we need to watch it too
   if (target.actionListener != null) target.addActionListener(queue);

   // now get the real toolkit to create the button
   return theToolkit.createButton(target);
}
```

Code Fragment 7.2: `createButton`

Notification of all application-relevant user actions will be sent to the `ReporterQueue`. The `Reporter` object will relay these reports to HERCULE. There is one more thing that

has to be done. HERCULE needs to know when `Windows` are being displayed on the user interface and when they are removed. To be informed about this, the `ReporterQueue` also listens to all window events, thus being informed about when windows are *shown* or *hidden* from the user interface. When this happens a *show component* or *hide component* report is generated and added to the queue.

The code given in Code Fragment 7.2 looks bound to work and indeed it does keep HERCULE informed. All listeners are structured as a linked list, the first of which is the application listener. Therefore the application will be notified first and be allowed to complete all execution which hinges on the event. Only then is HERCULE notified. This makes it impossible to provide immediate dynamic feedback with respect to the status of current application-server interaction, because the method-invocation reports will arrive long after all activity has been completed. To alleviate this, the order of the two listeners must be reversed. This is achieved by placing extra code within the `createButton` method of the `ProxyToolkit` as shown in Code Fragment 7.3.

```
if (target.actionListener != null) {
    // ok, there is a listener, remove it and
    // put the reporter queue in as the first listener
    // then add the old listener again
    java.awt.event.ActionListener listener = target.actionListener;
    target.removeActionListener(listener);
    target.addActionListener(queue);
    target.addActionListener(listener);
} // listeners registered
```

Code Fragment 7.3: Registering Interest in Events

This section has outlined the mechanisms used to record user activity at the user interface and to watch changes in displayed windows. Together with the previously defined internal structures representing these windows, the meaningful information can be provided about user interaction with the system.

7.2.5 Maintaining and using the internal image of the GUI

The *construction*, *status* and *event* reports generated by the `ProxyToolkit` are used to build up a tree structure, depicting the appearance of the user interface, as shown in Figure 7.6. HERCULE keeps track of user activity by maintaining a history of windows which are shown at the user interface. HERCULE also keeps track of user actions which cause a change in the user interface appearance. Event reports will keep the tracking program informed of all activity which will then be up to date with exactly what the user has been doing at any time, together with the effect on the user interface of that user activity [Ren99].

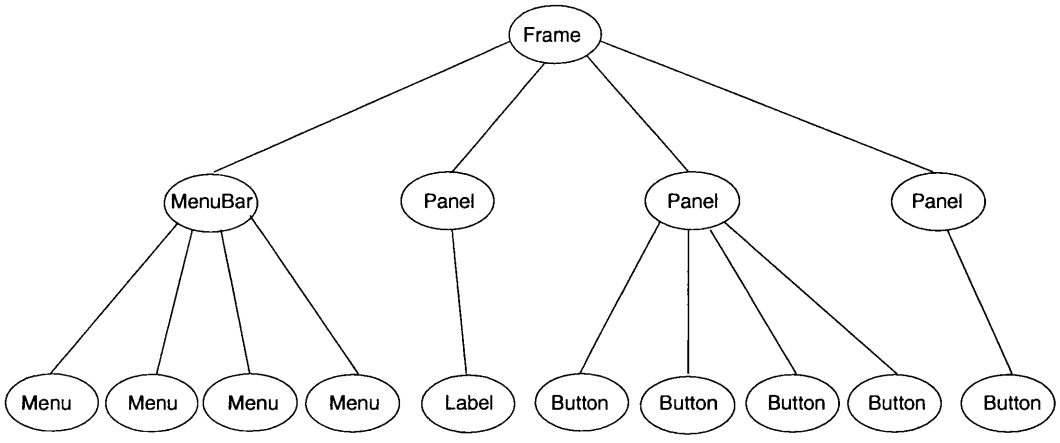

Figure 7.6: The Internal User Interface Representation

7.3 Observing Server Communication

This section discusses the Java-specific application of the minimal impact proxy pattern for observing communication with the server, with the proxy inserted between the client application and the rest of the CBS, positioned as shown in Figure 7.2, using Enterprise Java Beans (EJBs)[Tho98a] as server components. Although the mechanism has been developed specifically for CBSs using the *Java Naming and Directory Interface* (JNDI) [Mic98c] to access EJBs, it appears not impossible to customise for other communication models where a naming service is used to locate server components and components separate interfaces from implementation.

7.3.1 The Enterprise Java Beans Component Model

The EJB specification requires a client application to make use of the Java Naming and Directory Interface (JNDI) package to contact the application server. Each bean will have a JNDI name which is published by the server and which will be supplied by the application in order to enable JNDI to find the component. It will have two distinct interfaces, a `Home` interface (for managing bean instances) and a `Remote` interface (for business-logic methods). The object that implements the `Home` interface is called an `EJBHome` object, while the object implementing the `Remote` interface is called an `EJBObject`.

JNDI requires the client application to establish communication with the server housing the server components before any connection can be made with those components. The context must implement the `javax.naming.Context` interface. The `javax.naming.InitialContext` class implements the `Context` interface, providing the necessary context to the application. The JVM makes use of a `CLASSPATH` environment variable that can be exploited to ensure that the JVM loads a *proxy* class *instead* of the original class, simply by putting the location of the proxy class ahead of the location of the original class in the

CLASSPATH. The proxy for the `InitialContext` class will be dynamically inserted by making use of the above-mentioned CLASSPATH mechanism. This works because of the JVM's equivalence mechanism which considers two classes to be equivalent if they have the same name and are loaded by the same class loader. By giving the proxy class the same name one can guarantee that the JVM will accept it when the application requests that the class be loaded.

Consider an EJB which provides the functionality required to create new accounts, close existing accounts, withdraw funds or deposit funds. The EJB is called `accountBean`, which is supplied together with two interfaces, the home interface called `AccountHome` and the remote interface called the `Account` interface. The client application goes through the following steps to use an EJB:

1. Establish a starting point to link the application program to the available EJBs contained in the EJB server, as shown in Figure 7.7, by instantiating the `InitialContext` object. The `InitialContext` object needs some properties to identify the server to be contacted. The first, and most critical property, is the *Universal Resource Locator* (URL) which identifies the location of the EJB server. Other properties include the context factory (which will produce the required context object), the user login name and the password. The client program establishment of context is shown in Code Fragment 7.4.

```
// build up the properties of the connection
Properties properties = new Properties();

// put the URL, initial context factory, user name
// and password into properties

// get the initial context
Context theContext =  new InitialContext(properties);
```

Code Fragment 7.4: Application calls to establish initial context

   The `InitialContext` object implements the `Context` interface, and establishes a naming context. The context is an object whose state is a set of bindings with distinct atomic names. Since, in this case, we are "pointing" the context at the URL of the EJB server, this `Context` object will allow us to obtain a link to any EJBs residing in the EJB server.

2. Get the `EJBHome` object, as shown in Figure 7.8, by calling the `lookup` method in `InitialContext`. The client program requests the home object by providing the JNDI name of the EJB (`accounts.accountBean`) and invoking the `InitialContext` `lookup` method as follows:

```
AccountHome home = (AccountHome) theContext.lookup("accounts.accountBean");
```

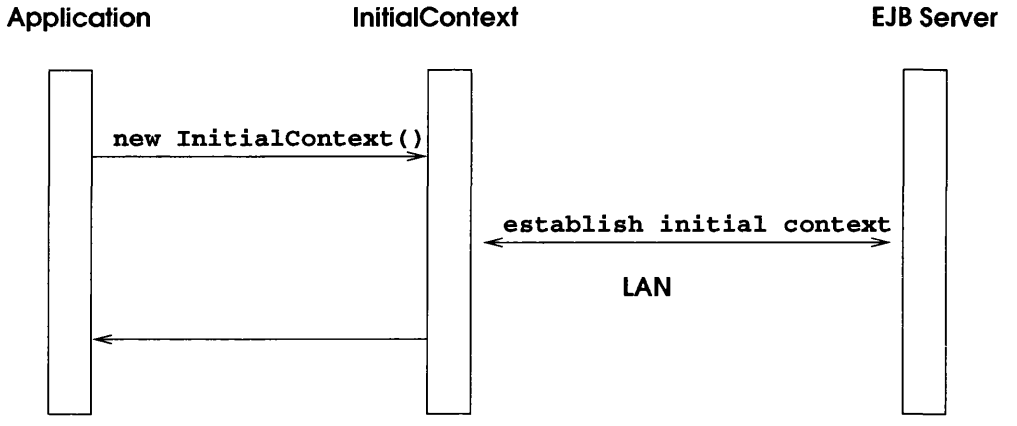

Figure 7.7: Establishing contact with the server

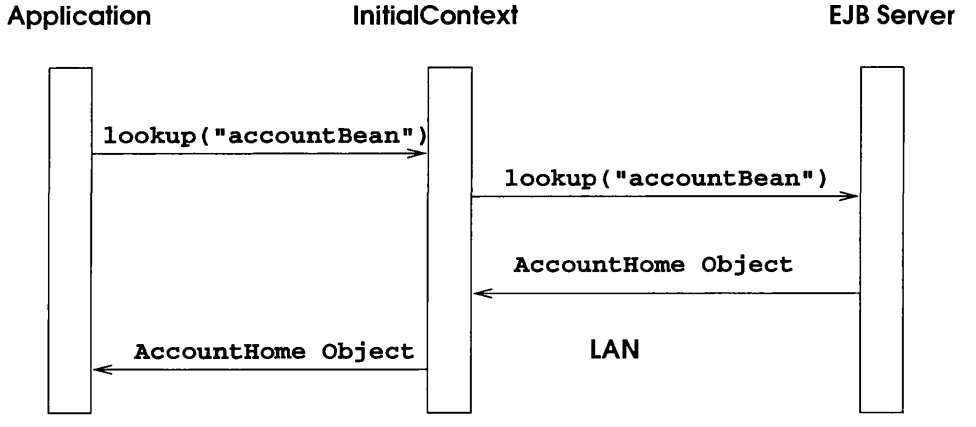

Figure 7.8: Getting the Home Interface Object

The home object implements the `AccountHome` interface and will be used to locate existing beans, or to create new beans.

3. Use the `EJBHome` object, as shown in Figure 7.9, to get instances of individual `EJBObjects`, each of which is identified by means of a key object. This could be done as follows:

```
currentAccount = (Account) home.create(accountKey);
```

The `currentAccount` `EJBObject` implements the `Account` interface.

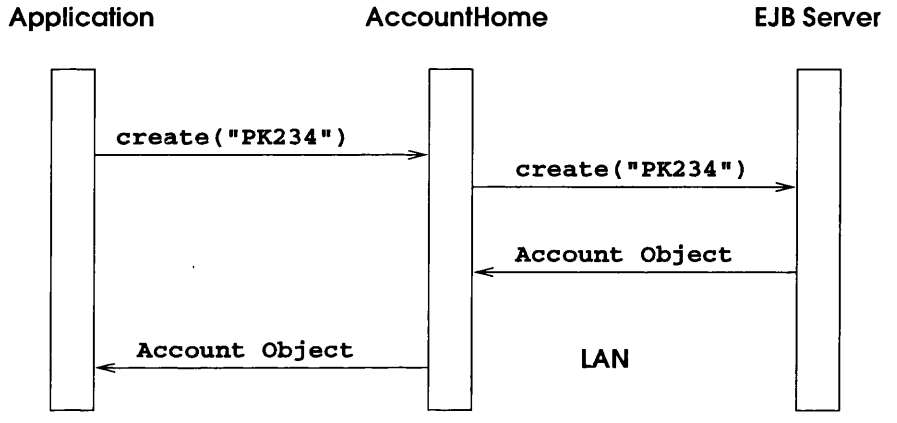

Figure 7.9: Getting the EJB Object

7.3.2 Using Proxies to Intercept Communication

There are two steps involved in tracking all application interaction with the EJB server. The first is to insert proxies at each of these three communication stages. The next step requires the reports generated by these proxies (`MiReports` — Method Invocation Reports) to be forwarded to HERCULE.

7.3.2.1 Inserting the Proxies

To insert a proxy at the connection stage, the system has to generate a proxy which will implement the `Context` interface, in the same way as is achieved by the `InitialContext` class — since the application's source is not going to be altered in any way. This proxy `Context` object has been specially developed, but will now serve to insert proxies into an application using the `InitialContext` class to establish an initial link to a middle-tier server. The proxies for the `EJBHome` and `EJBObjects`, on the other hand, will have to be uniquely generated for each different EJB. In order to ease this process, HERCULE generates the proxies automatically by using the class files and reflection. The tool provided for generating these proxies, as part of the prototype implementation, is discussed in Section 7.4.1. To explain exactly how the proxies are engaged at runtime:

1. In the first place, HERCULE needs to intercept calls to the `InitialContext`. The `Context` interface is therefore implemented and also named `javax.naming.InitialContext`. This class is put into a location which was inserted into the `CLASSPATH` *ahead* of the original `InitialContext`, thus ensuring that the JVM loads the *proxy* `InitialContext` and not the original one. When the application instantiates `InitialContext`, the *proxy* implementation of `InitialContext` is called. A special context, `ProxyContext`, is now instantiated (this also implements the `Context` Interface) and this instance is returned to the application. Since the application is expecting an object that implements the `Context` interface, it is unaware of the substitution.

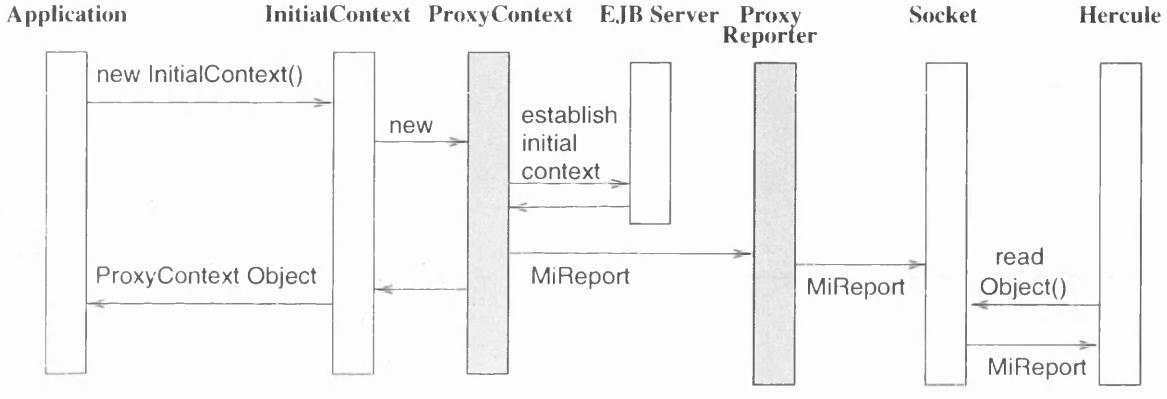

Figure 7.10: Establishing contact with the server using a proxy

All communication between the application and the EJB server is now routed through this `ProxyContext`. This object holds a reference to the actual `InitialContext`, allowing it to observe all calls made via this object to the server. The procedure is illustrated in Figure 7.10.

2. When the application makes a call to the `InitialContext` to request an object that implements the home interface, the `ProxyContext` instantiates a *proxy* implementation of the home interface (`AccountHomeProxy`). The required `EJBHome` object, implementing the home interface, is requested from the server and the `AccountHomeProxy` object is given a reference to this object. The instance of `AccountHomeProxy` is returned to the client. Once again the client application is none the wiser, since the proxy also implements the `Home` interface. The proxy relays all calls to the actual `EJBHome` object and returns replies to the application. See Figure 7.11.

3. When the application makes a call to the `EJBHome` object to request a specific bean, the `AccountHomeProxy` object instantiates a proxy `EJBObject` (`AccountProxy`). The `EJBObject` implementing the `Account` interface is requested from the server, and the `AccountProxy` is given a reference to this object. It then acts as a channel through which all calls are relayed. The interception is illustrated in Figure 7.12.

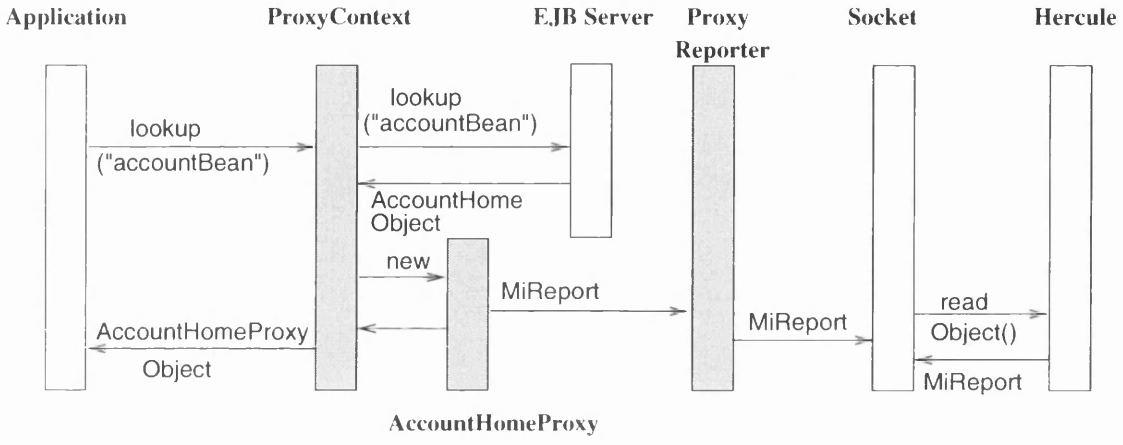

Figure 7.11: Getting the EJBHome Object using a proxy

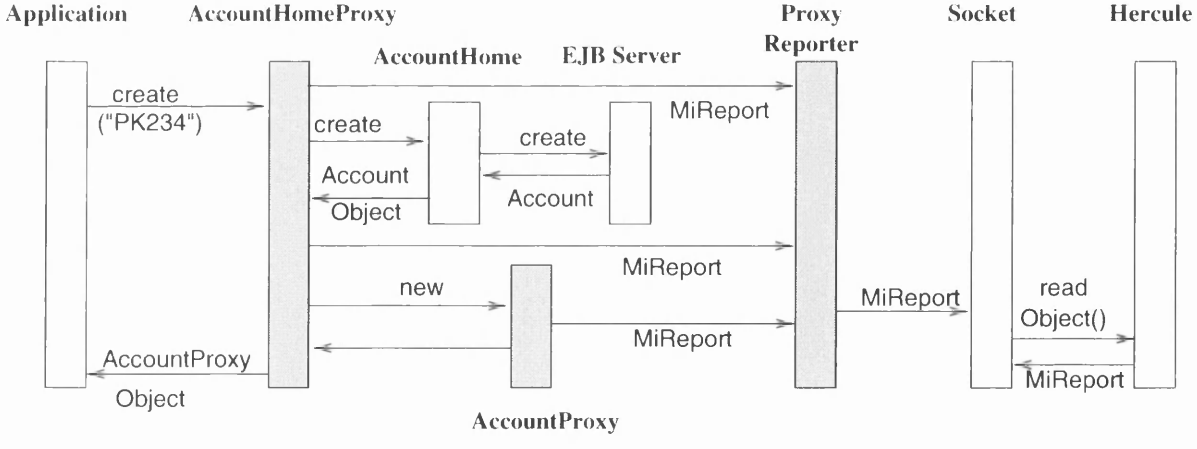

Figure 7.12: Getting the EJB Object using a proxy

This explains how to insert proxies at *each* level of the communication with the server, and these proxies then generate reports by means of which all communication with the server is monitored.

7.3.2.2 Sending the reports to HERCULE

The server proxies also need a structure which will facilitate the sending of reports to HERCULE. The `ProxyReporter` provides for this. The proxy object, both `EJBHome` objects and `EJBObjects`, will essentially have to report on every method invocation, giving information about the method, the parameters supplied, and the time the invocation occurred. Once the method has been executed, the proxy will either report on the successful completion of the method invocation — reporting the return value if there is one — or give details about the exception thrown, in the case of an error. The `ProxyReporter` receives these reports, and uses a `Socket` connection to relay them to HERCULE.

7.3.3 Using the reports generated by the proxies

When HERCULE receives the reports, they have to be stored so that the information can be retrieved at any time for feedback purposes. It is important to realise that the server proxies are totally unaware of the user interface proxy and that they therefore have no communication with one another. The only way that HERCULE can link user actions to server method invocations is by using the time factor enclosed within the generated reports. Therefore, when server reports are received, these actions will be linked to the user actions which preceded them.

When storing the proxy information (derived both from the user interface and the server), it is vital to store it in the form of Episodes. This is necessary because the user activity must be linked to system actions so that a link is established which can be exploited by the display mechanism to portray the application activity to the user. It is still necessary to keep them apart for some specialised feedback requirements, so HERCULE will store a list of UA-sequences and link each UA-sequence to the MI-sequence precipitated by the UA-sequence. These two lists will be linked one to the other, forming a history of session Episodes.

7.4 The Descriptor Tool and Proxy Generator

HERCULE has two distinct phases of use: discovery and runtime. The discovery phase is a *customisation* phase, which serves to inform HERCULE, essentially providing a generic feedback mechanism, of the server components which will be used by an application. During the runtime phase, the results of the customisation will be used to facilitate the required feedback.

Since the programmer has to generate the proxies and the descriptors at least once, to customise HERCULE, the two tools have been merged, as shown in Figure 7.13. The following sections will discuss the implementation of these tools.

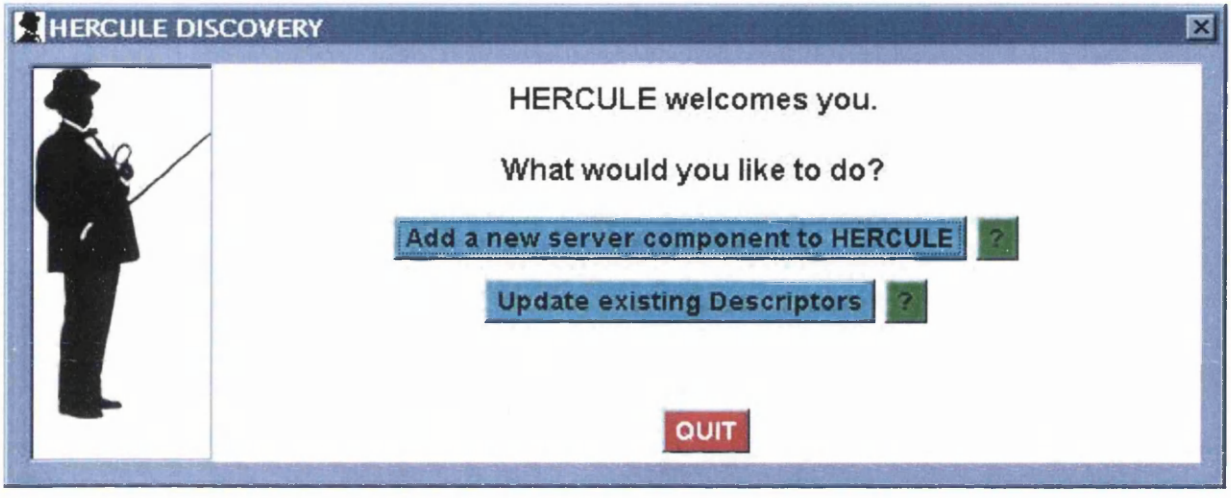

Figure 7.13: Customising HERCULE

7.4.1 The Descriptor Tool

This "discovery" phase is executed prior to HERCULE being used, and as often as necessary after that as the programmer becomes more familiar with the operation of the component. HERCULE makes use of the server component documentation to customise the framework for a particular server component. In Section 6.3, three documents were mentioned that *have* to be provided together with a server component:

1. An *Application Programmer Interface* (API) document, which explains the purpose of the component and gives details of method functionality. Examples of such documents are those found as `javadoc` [Mic98b] output.

2. One or more interface classes through which the component can be accessed.

3. A deployment document which specifies the context dependencies of the server component and explains how the component should be deployed.

Many component vendors will choose to provide far more, but HERCULE only relies on the basic minimum being provided. The delivered documentation is "mined" in order to extract *descriptor* objects that hold details about the methods used to access the server components, and to generate proxies.

Descriptor objects are essential to the visualisation of session activity. Tracking will only be meaningful if its results can be depicted in an information-rich and useful fashion. In

order to provide the users with explanations of server activity, the method invocations should be described in terms easily understood by the user, rather than in language familiar to the programmer of the system. These explanations are all to be found in the server component API documentation and the descriptor objects can thus be derived from these documents. Since Java class documentation is generally produced by `javadoc`, this makes the mining process simpler[4]. This mining process should produce at least an *adequate* descriptor object, since it contains the information as obtained from the API document. In order to improve this object, HERCULE provides a tool to allow the programmer to augment the descriptor object. With the programmer's assistance the descriptor object can be augmented to make it even more helpful to the end-user.

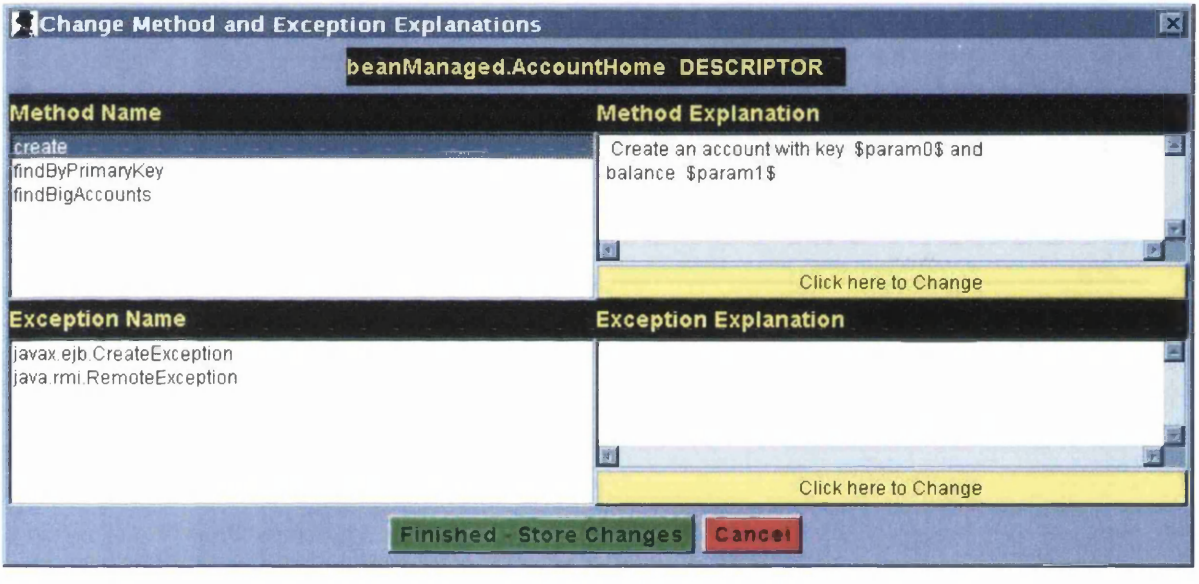

Figure 7.14: Permitting the Programmer to Augment Descriptors

Parameters used in method invocations can be inserted into the explanations of these methods. This will allow the programmer to customise the explanations of method invocations and exceptions thrown by the methods, according to the parameters provided by that particular invocation.

[4]If this is not done by `javadoc`, it becomes more difficult to mine since we have no idea how the documentation would be structured. The next EJB specification requires the use of *Extensible Markup Language* (XML) for this documentation, which would make the process even simpler because we no longer have to rely on the vagaries of the `html` being produced. This could possibly change from one version of `javadoc` to another, which would invalidate the current generation code. XML is easily parsed and does not suffer from these limitations.

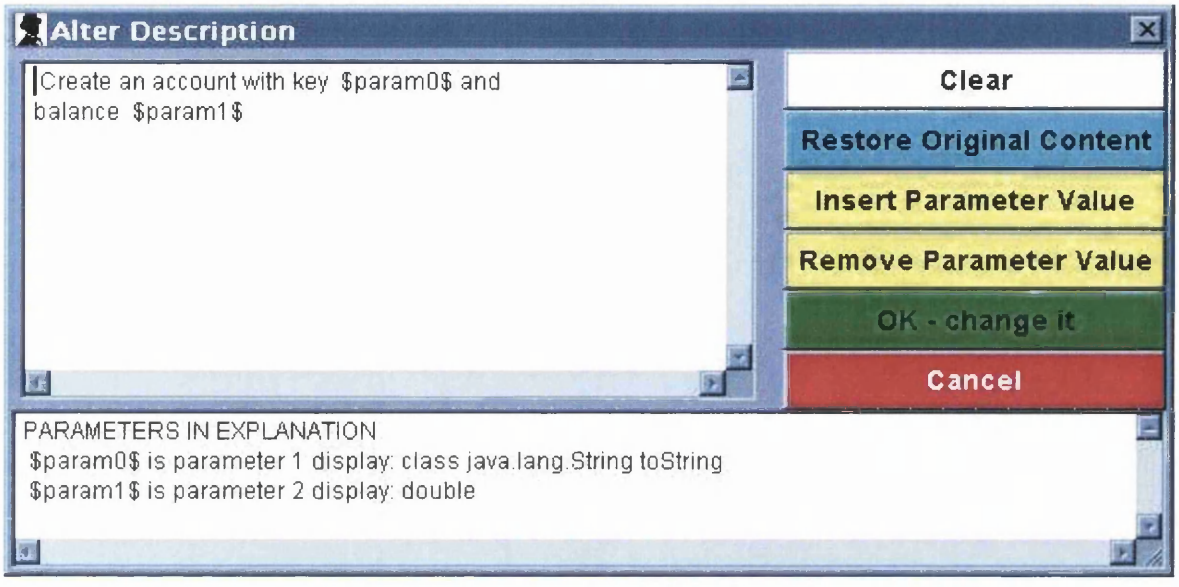

Figure 7.15: Changing a Method Explanation

7.4.2 The Proxy Generator

The proxies conform to the *wrapper* or *decorator* pattern [GHJV94]. This is one approach to adding reflection to statically typed languages [WS99]. Some examples can be seen in the work of Karaorman *et al.* and De Oliveira Guimaraes [KHB99, De 98]. Many implementations of reflective[5] Java rely on customised JVMs or require access to the source code of the application — examples are cited by Welch and Stroud in [WS99]. Since one of our design decisions stems from a strong desire to be non-invasive and optional, neither of these options is attractive.

If we want to engage proxies using a standard platform, without changing the source code, there are two ways to go about it. One is to make use of byte code transformations at runtime, while the other is to generate proxies offline and insinuate them into the system by manipulation of the `CLASSPATH` at JVM runtime.

The first mechanism has been applied successfully by the *Dalang* prototype and its extension *Kava* [WS99]. However, the approach taken in these projects is aimed at implementing meta-object protocols for commercial off-the-shelf components, whereas the focus here is on reporting on the activities of specific middle tier components. Whereas the changing nature of meta-object protocols[6] will make it feasible to re-generate and compile wrappers with each program execution, in the case of reporting, the requirements are stable. It is wasteful

[5]The `java.lang.reflect` package is wrongly named, since it allows introspection, but not actual reflection. Reflection implies the ability to change the behaviour at runtime — and the `java.lang.reflect` package does not allow that.

[6]Meta-object protocols allow the runtime insertion of additional behaviour into a system. This could cater for non-functional requirements such as distribution or concurrency, for example.

to generate and compile the proxy classes for each execution when it can be done once and repeatedly re-used thereafter.

The other concern is that the runtime compilation of wrappers may affect the performance negatively. The final factor which swayed the decision in favour of the second option was the sheer simplicity of the approach — the byte code transformation approach is intricate and admirable, but rather complex. It is also not clear whether the specialised class loaders written to expedite this scheme would work for future JDK releases.

The generation of the proxies was made possible by the introspective abilities of the Java language i.e. the `java.lang.reflect` package [Mic99]. This package reveals information about the interfaces needed to generate wrappers — method signatures and inheritance details and so on.

The proxies have the same methods as the interfaces. Methods will be invoked on the proxies by the application, the proxy will invoke the methods on the actual stub object, receive the reply, and pass that back to the application. In order to carry out its task, the proxy will report to the communication agent before the method is invoked on the stub, and after the reply has been received from the stub, before passing it back to the application.

7.5 The Runtime Feedback Tool

HERCULE tracks application activity by dynamically inserting proxies, and extracts information based on the reports generated by these proxies. HERCULE operates based on two types of inputs. The first is made up of the documentation and Java class files delivered with the EJB. The second comprises the reports generated, at runtime, by the proxies. HERCULE must use the information from this documentation to customise itself. This customisation facilitates the operation of the proxies at runtime. HERCULE receives two types of reports from run-time invoked proxies:

1. *User interface reports*: signaling events and the user interface construction. These events enable HERCULE to keep a history of user interface appearance and user activity.

2. *Middle-tier component method invocation reports*: The reports received here indicate different *stages* of server component communication:

   (a) *Contact*: initial establishment of communication with the server;

   (b) *New Server Component*: initiation of a new interface object;

   (c) *Interface Object Activity*: method invocations on the interface object;

HERCULE runs in a separate process so that its execution and termination are not dependent on the application. When HERCULE executes, it is initially in an inactive mode while it waits for the application proxies to make contact. Upon receiving the first report, which

informs HERCULE that the application is up and running, and that the middle-tier server was contacted successfully, HERCULE enters feedback mode. In feedback mode HERCULE receives messages about application activity and provides feedback to the user.

HERCULE registers the termination of the application by the cessation of the `Socket` connection either from the server proxies or the GUI proxy. HERCULE stays active so that the user can use the display to provide post-execution feedback. This will be particularly useful if the application terminated erroneously or if the user needs to confirm actions taken during the session. It also allows the user to summon help if something has gone wrong and enables the user to demonstrate the actions taken, should a support person be summoned for assistance.

7.6 Application Activity Visualisation

Once the UA-sequences have been linked to the MI-sequences and the Episodes have been constructed, the results need to be depicted in a helpful manner on the screen. There are many aspects of this interaction that could be depicted, but for HERCULE, the decision was made to depict the success or failure of each Episode. This decision was made because the focus is to provide end-user feedback and the success or failure of an Episode is of critical interest to the end-user. Since a particular application session could easily generate many Episodes, the display chosen has some important characteristics:

- It should be able to depict either one or many Episodes in a clear manner, so that the user can obtain as much information as possible at a glance.
- It should not intrude, but offer the user assistance.
- It should allow the user to step backwards in time to view and confirm previous actions.

7.6.1 Characteristics of Visualisation

Section 6.5.2 suggested that the following feedback should be provided:

- A status display.
- A current time display.
- Explanations of latest episode — tailored to the current user role.
- Access to reconstruction of context.
- Summary information — such as, for example, a graphical display indicating the performance of the network.
- An overview of episodes — a display offering the "overview and zoom" facility, which will allow users to choose which Episode to access.

- An expanding facility — linked to the above feature, giving extra information about the chosen Episode.

The display designed for HERCULE was created with those requirements in mind and satisfies them as follows:

- It provides a mechanism to enable the user to get information about all of the Episodes for the entire application.
- It allows detailed information about Episodes to be obtained quickly and easily.
- It does not intrude, but is always available as an icon, offering the possibility of obtaining feedback at any time.
- It allows the user to obtain information about previous Episodes quickly and easily.

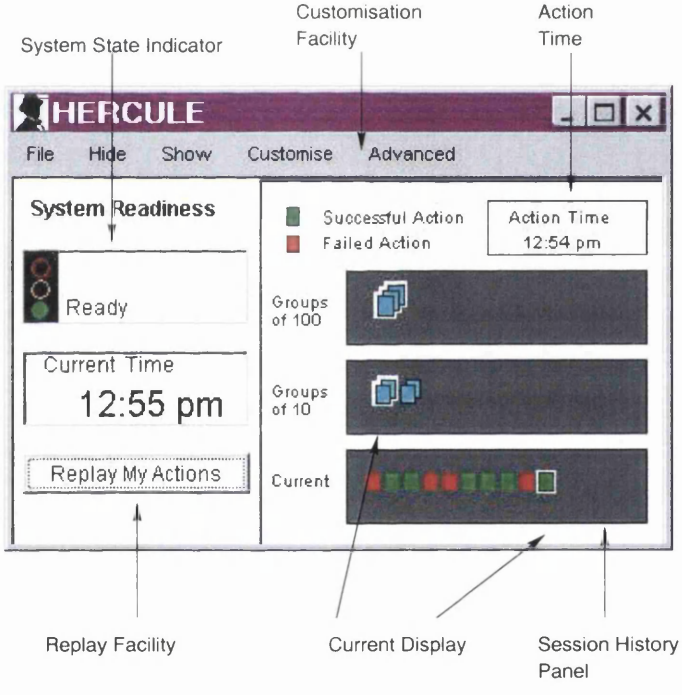

Figure 7.16: The HERCULE Display

The icon chosen for HERCULE is that of a man's head, shown in black on white. This has been chosen so that the user can easily identify the HERCULE window and the icon, should it be minimised.

7.6.2 Interactivity of the Display

At runtime, the HERCULE display (Figure 7.16) provides the following information, which is dynamically updated as the user works:

1. A traffic lights widget depicts the current system state. This will display:

   - *red* when the application cannot be tracked. The legend beside the traffic light will display the result of HERCULE's attempt to diagnose the cause. This could be: due to a server breakdown or a network problem; or because the application has not yet started executing; or because the application has terminated;
   - *orange* when the middle tier server is busy servicing a request; and
   - *green* when HERCULE is waiting for application activity. Since humans are so much slower than computers, one can expect the display to be in this state for a great percentage of the time — reflecting the time spent by the user assimilating the screen display and deciding what to do next. HERCULE will depict activity once the user has provided inputs and signaled that they should be processed, otherwise it simply waits.

   The traffic lights display is a universal symbol, and adequately sends the required message in most cultures.

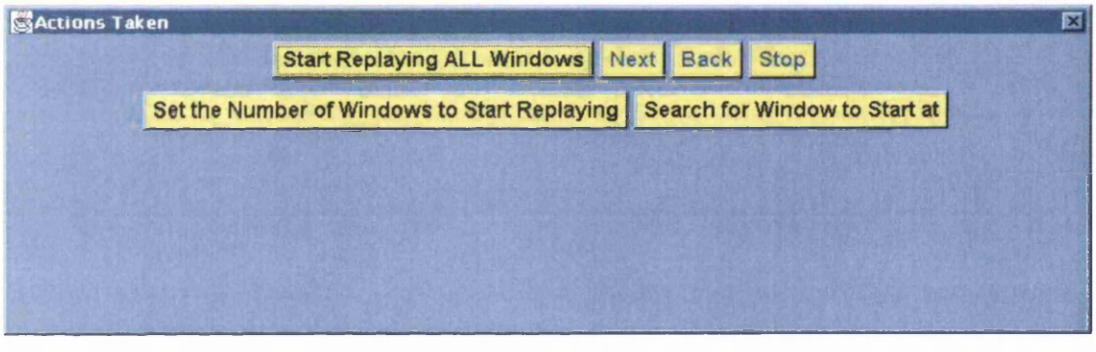

Figure 7.17: The Playback Facility

2. A `Replay My Actions` button will summon a playback facility, shown in Figure 7.17, which allows the user to view a screen replay of all UA-sequences as they took place. This shows the windows displayed by the application to the user, one at a time. The user can control the transition to the next window by clicking the mouse, and so control the pace. Each window will highlight the action which caused the transition to the next window. For example, if the user clicked on a button, that button would be highlighted by setting the background colour to yellow in the replay window.

   To allow extra flexibility, the user can search for a particular window with a key phrase in it, step back a certain number of windows or simply replay all activity from beginning to end.

   By providing this functionality, HERCULE supports users by alleviating their weaknesses (such as limited working memory), while capitalising on and utilising their

strengths (such as swift pattern recognition, and the ability to retrieve relevant information about the meaning of these patterns quickly). The replay mechanism has no effect on the application whatsoever, in accordance with the non-intrusion policy, and should be considered to be rather like an action replay used in television sports broadcasts.

3. A session history panel which presents all Episodes hierarchically, displayed in three separate panels:

   - the bottom panel displaying the *last ten* Episodes;
   - the middle panel depicting *groups of ten Episodes*; and
   - the top panel depicting *groups of hundreds of Episodes.*

   Each distinct Episode is displayed as a coloured rectangle. This depicts the result of the *MI-sequence* resulting from the Episode UA-sequence as:

   - *red* if it failed — assumed if the server throws an exception,
   - *yellow* if the outcome is pending, and
   - *green* if it succeeded — assumed by the absence of an exception.

   The colour red is traditionally used in the western world to indicate either danger, or heat, while green is used to signal safety [Tra91]. The use of these is highly culture-specific since the Chinese traditionally use green to symbolise death, with red symbolising luck and good fortune [War00]. The link of the colour to the meaning is shown in the legend at the top of the session history panel, so that this type of confusion can be avoided. The best option would be to allow users to choose the colours themselves, but this would add to the complexity of the display, something which should be avoided. These colours are used so that an error will automatically "pop out" of the background, so that the user will be more likely to notice it.

7.6.3 Extensibility of the Display

The HERCULE display is dynamically extensible, so that the identification of a new user feedback need can be accommodated. New HERCULE feedback components can be coded, and added to the HERCULE display at runtime. The top section of the display, as shown in Figure 7.21, will always be displayed, since it provides the core functionality of the display. A programmer can add a new feedback component, by coding a class which *must* extend the `HerculeComponent` class. The inheritance hierarchy for a HERCULE feedback component is shown in Figure 7.19.

The component could implement either the `HistoryListener` or the `OutcomeListener` interfaces, or both, depending on the notifications required. To add the component to the

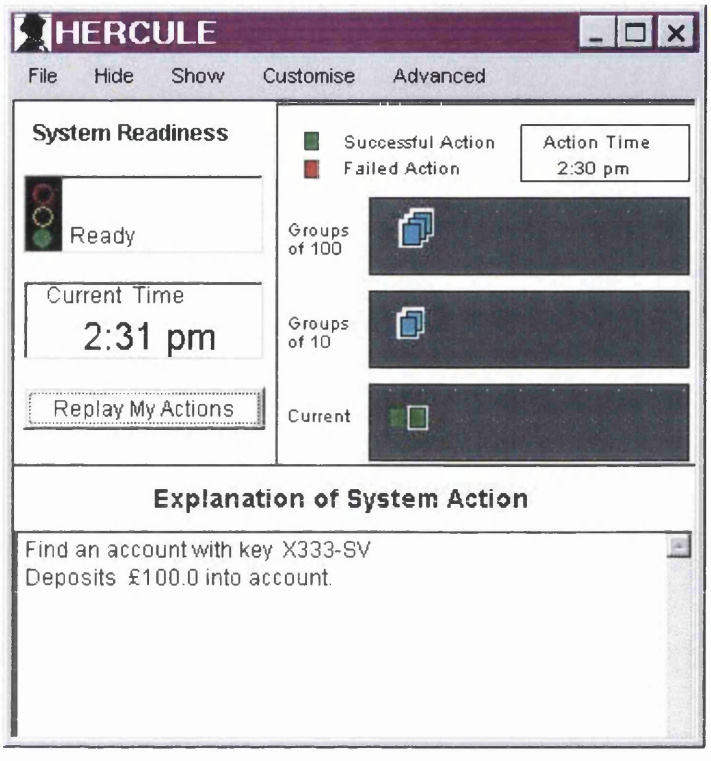

Figure 7.18: The User Viewing an Explanation of a Previous Episode

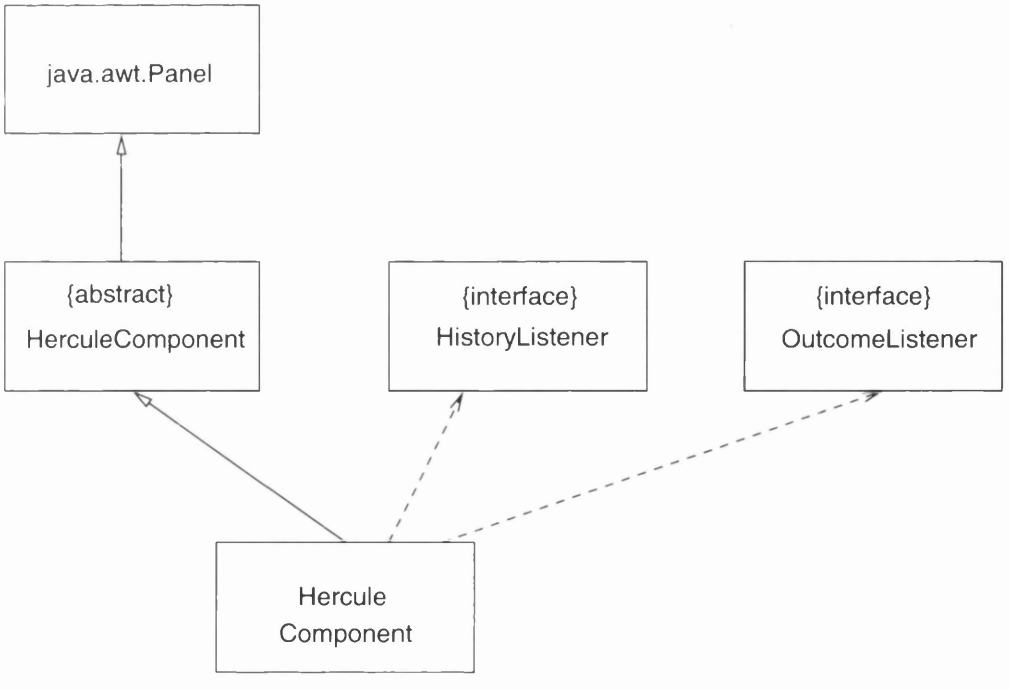

Figure 7.19: Structure for extending the HERCULE Display

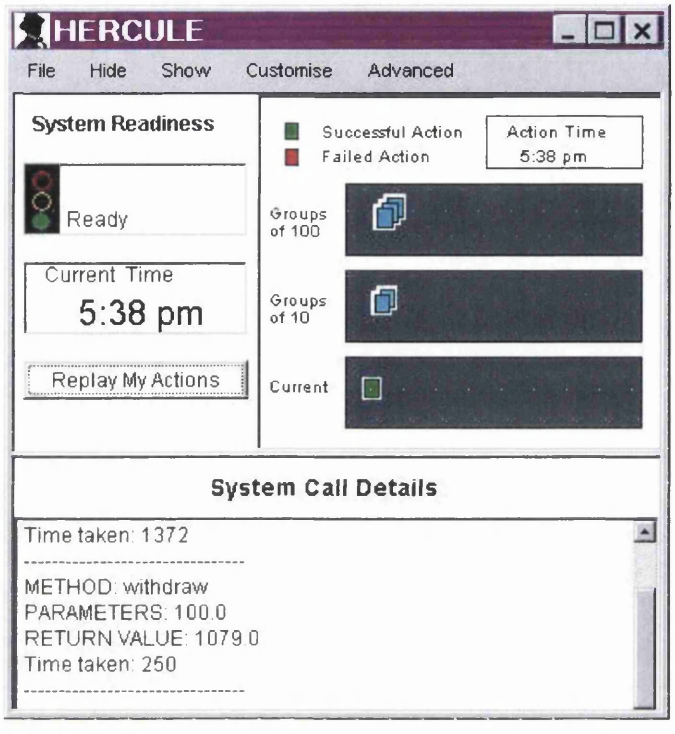

Figure 7.20: The Extended HERCULE Console

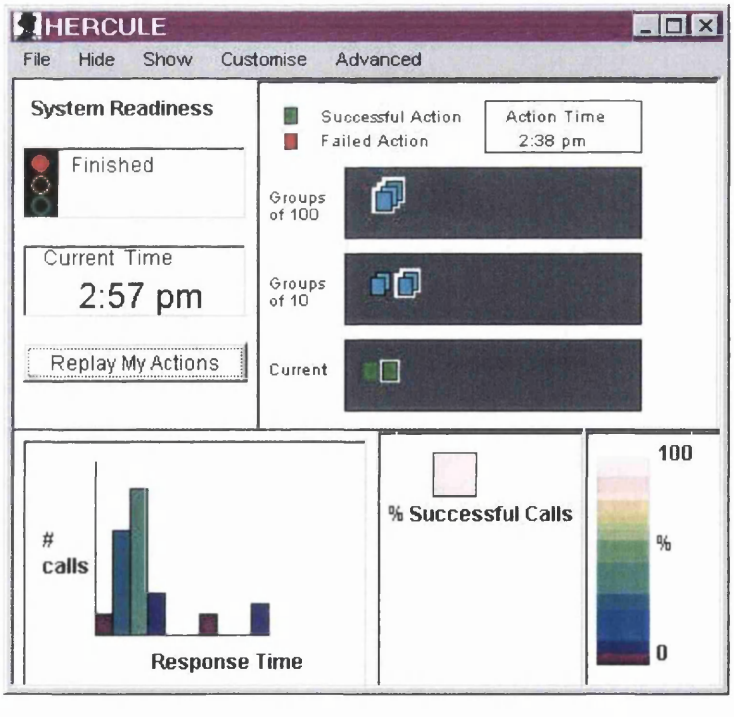

Figure 7.21: The HERCULE Console showing the Support Panel

points to the Episode for which feedback is currently being given in the visible feedback components. The *Current Display* label in Figure 7.16 points to the highlighted rectangles in the history panels, indicating that the most recent Episode MI-sequence explanations would be displayed (if any feedback components were visible).

The user's immediate feedback requirements with respect to individual Episodes, as indicated by clicking on a block which represents a previous Episode, will be met by dynamically reflecting the feedback for that Episode in the information displayed by each of the visible feedback components. On the console shown in Figure 7.18, the Episode actions are explained as `Deposits £100.0 into account`. This is not the explanation of the most recent Episode MI-sequence, since the highlighted rectangle is in the last but one position, indicating that the explanation belongs to the second last Episode.

7.7 Conclusion

The implementation described in this chapter has produced a prototype of the HERCULE feedback enhancing framework. It was mentioned in Chapter 5 that the general concept of such a framework required the use of a language with introspective qualities. It should be clear from the discussion in this chapter that these were indeed used extensively throughout this project. The implementation was done using JDK 1.1.7, because the middle tier server used that version, and the 1.2 version was not available when implementation commenced.

The scheme for engaging the proxies makes use of manipulation of the `CLASSPATH` in order to insert the proxies into the system. This feature is used rather differently in JDK 1.2. Preliminary tests have indicated that the hook provided by JDK 1.1.7 which facilitates the insertion of the user interface proxy is still provided in 1.2. The only difference is that JDK 1.2 expects to find any class with a name starting with `java`, in a special place — i.e. within the `rt.jar` file provided by Sun. Since the `ProxyToolkit` *must* be part of the `java.awt` package because all the methods in the abstract `Toolkit` class are `protected` and cannot be invoked by a member of another package, the only way to make use of the user interface proxy proposed in this chapter is to add the `ProxyToolkit` class file to the JDK `rt.jar` file. This is trivial and, although it could be considered to violate the non-intrusiveness aim, it does not do so to an unacceptable extent.

Since the `JNDI` package is external, or additional, to the core JDK, the `CLASSPATH` facility is used by the JVM to locate it, which means that the mechanism explained in this chapter can be used without alterations.

An alternative to the `CLASSPATH` mechanism is the use of a specialised class loader to insert the proxies. This classloader can detect members of specific classes, and substitute the proxy classes as required. This mechanism has been used by Welch and Stroud [WS99] in developing their *Kava* byte code transformation approach.

This chapter discussed the implementation of the HERCULE prototype. The following part of this dissertation will evaluate the software and draw the final conclusions.

part V

Epilogue

It was the best of times, it was the worst of times.

Charles Dickens. A Tale of Two Cities. 1890

Reason, or the ratio of all we have already known, is not the same that it shall be when we know more.

William Blake. 1788

Basic research is what I'm doing
when I don't know what I'm doing.
Wernher Von Braun

We have a habit in writing articles published in scientific journals to make the work as finished as possible, to cover up all the tracks, to not worry about the blind alleys or describe how you had the wrong idea at first, and so on. So there isn't any place to publish, in a dignified manner, what you actually did in order to get to do the work.
Richard Feynman

chapter 8

Evaluation

The evaluation of HERCULE proved to be the most difficult part of the research. Since this tool is so unlike other software development tools there is no obvious way of evaluating it. Even for standard tools, no widely accepted systematic assessment method exists [CMH92]. If one is to prove the value of HERCULE conclusively, there are various aspects of HERCULE's use that should be evaluated. Evaluation is often done in a laboratory, and the resulting findings are essentially based on short-term user experience of the tool. Laboratory-based short-term evaluation is not the best way of evaluating HERCULE because it has the following shortcomings:

- Evaluation involving the use of HERCULE by a number of end-users would be the ideal way to evaluate the HERCULE display. The need for HERCULE is deemed to be greatest in complex systems, in which people are forced to learn how to use the system in order to perform their duties. People who *must* use a piece of software for

some reason are motivated enough to overcome problems in understanding the system and will simply *have* to master it. Volunteers are not motivated by this need and it is unrealistic to expect it of them. It is therefore difficult to set up an experiment which tests the efficacy of HERCULE within a short experimental period of a half hour, or an hour, with a volunteer group. The application must be kept simple if the user is to have any chance of making use of it within the short period and the very simplicity of an application which can be used in an experimental setting makes HERCULE somewhat superfluous;

- Subjects in such an evaluation are subject to the *Hawthorne* effect, the tendency for individuals to respond positively to special attention or a change in routine [Mil94]. Any increase in productivity and performance, or perceived ease of use, can therefore not be linked conclusively to HERCULE;

- One has to rely on the subjects' subjective evaluation of their workload, performance and satisfaction with respect to the use of the tool. Studies suggest that users often do not report effects that they plainly do experience, either because of a sense of pity towards the developer of the tool, or a sense of irritation with the entire evaluation process, or because they find it difficult to evaluate their experiences effectively [WS00].

- Finally, there is always the need to test a tool such as HERCULE in a real life setting rather than in a laboratory in which the findings are not necessarily applicable to authentic work situations [And90].

The alternative to short-term evaluation is to test HERCULE over the long term in an industrial setting, so that the long-term benefits can be assessed. Aspects to be evaluated would include the following:

1. Chapter 3 has derived a classification of disruptive events and motivates the need for user assistance in recovering from such events. It is necessary to confirm the correctness of the classifications of each of the separate quirks — error, interruptions and breakdowns. It would also be useful to determine the cumulative effect of these events on users' working day. While other researchers have studied these concepts in isolation, a study which considers all events together could be interesting and would either validate, or suggest changes to, the derived classifications;

2. Easing the process of recovery from interruptions in particular is a subject that has not received much attention from researchers. It is hoped that HERCULE will assist users in recovering from interruptions quickly and with little effort, by reminding them of past actions, thereby easing recovery of context. In order to prove that this is indeed the case, it is necessary to observe users recovering from interruptions with and without HERCULE, and to time the recovery time. Previous studies suggest an unassisted

context recovery time of up to 15 minutes [vSBvL98]. A quantitative evaluation would be able to show whether this time is reduced by users using HERCULE;

3. The sociological impact of HERCULE is the aspect that will be most interesting and relevant to evaluate. Many people are intimidated by their computers. Other veer towards hate — and many become increasingly stressed as they attempt to use their computer to carry out essential tasks during their working day. It is hoped that HERCULE would have the effect of reducing these negative emotions and increasing general confidence in the computer. Feelings with respect to particular applications in particular and computers in general can not be expected to change in the short term and any change in attitude would have to be gauged over a period of time. In determining the effect of HERCULE it is important for the evaluator to establish an amicable relationship with the user so that the user feels free to express displeasure or delight without fearing disapprobation. This too, is not to be hurried, since relationships take time to build up;

4. It is hoped that HERCULE will be of assistance to application programmers. The first obstacle in evaluating this is in overcoming the programmers' initial reluctance to use the tool, and then in gauging their reaction to it, and ascertaining whether HERCULE is indeed easing their task. One could rely to a certain extent on a subjective evaluation since programmers may feel positive enough about the tool to rate it highly. However, any subjective analysis is bound to be error-prone and the best test of HERCULE would probably come from an observable increased reliance on HERCULE during the system development process and in consequent suggestions from programmers about useful extensions to the tool.

As a consequence of the above factors, it was concluded that the long-term evaluation of HERCULE should be cited as a topic for future investigation. A short-term evaluation was carried out, in spite of its shortcomings, since any findings as a result of this evaluation would be helpful in obtaining an initial impression of the reception accorded to HERCULE by end-users and programmers. Different approaches to evaluation are discussed in the following section. The motivation for the preliminary evaluation methods chosen for short-term evaluation of HERCULE are also given. Section 8.2 discusses the results of the preliminary evaluation. Section 8.3 concludes.

8.1 Current Approaches to Evaluation of Tools

McKirdy and Gray [MG00] point out that many tools are chosen based on marketing material, journal reviews or word-of-mouth rather than by the use of evaluation tools. Evaluation methods have been proposed for some classes of tools, such as development environments, user interface development tools or *CASE* tools. For example, Mosley [Mos92] has developed

a five-step method to assist developers in selecting *CASE* tools. Her approach evaluates the proposed tool according to: ease of use, power, robustness, functionality, ease of insertion and quality of support. The tool is given a score, which indicates how well it measures up in each category. Mosley emphasises that the evaluation of a tool is only the tip of the iceberg and that the use of the tool in the organisation is a much bigger issue.

McKirdy and Gray [MG00] introduce their S.U.I.T framework, which can be used to evaluate the suitability of user interface development tools. They evaluate the tool according to categories which take human resource and organisational context into account. S.U.I.T also considers the ease with which the tool can be integrated into the existing working practice.

Poston and Sexton [PS92] propose that software tools be evaluated according to various criteria including productivity gain, quality gain, organisational changes required, platform changes required, functionality, response time, user friendliness and reliability.

HERCULE is the only tool that is specifically designed to assist an application programmer in providing feedback to the end-user. The agent implemented by Rich and Sidner [RS97], described in Section 5.5.3, is the only other tool the author has located which does something similar, although their tool requires the application programmer to provide hooks, which is not required by HERCULE. There can therefore be no comparison with other tools.

It is important to note that the evaluation of HERCULE should also be initiated from the perspective of the end-product produced by the software development process. Other software development tools will be used exclusively by the programmer and, while it might be easier to produce the end-product — a working application, the end-user will not have any interest in, or knowledge of, the tools used to produce the software. HERCULE is somewhat different, since the end-user will gain a direct and visible benefit from the programmer's use of HERCULE during the software development life-cycle — to whit, the HERCULE feedback window. Thus it is necessary to extend and modify the traditional evaluation criteria to include evaluation of the end product by the end-user in the evaluation process.

8.2 Preliminary Evaluation Results

For the purpose of a preliminary evaluation it was decided that HERCULE would be evaluated from two distinct perspectives. Firstly in terms of how the end-user (in any of a number of roles) perceives and uses the HERCULE display. The second perspective is that of the programmer. Evaluation here must assess the impact of the HERCULE facility on their task and determine whether it helps or hinders. Some relevant evaluation criteria have been selected from those proposed by Mosley [Mos92] and Poston & Sexton [PS92]:

1. *End-user assistance* — encompassing criteria such as functionality of the HERCULE display and the quality gain (with respect to feedback). Section 8.2.1 will discuss the evaluation of HERCULE from the end-user's perspective.

2. *Software development* — encompassing criteria such as ease of use, robustness, functionality, ease of insertion, productivity gain, organisational changes required and reliability. Section 8.2.2 will discuss how the HERCULE framework can be used by the application programmer. The evaluation of HERCULE by application programmers will also be described.

3. *Performance impact* — evaluation of HERCULE in terms of its effect on application performance and robustness. Since this did not fit neatly into either of the above categories, Section 8.2.3 will describe the results of the performance evaluation.

8.2.1 User Needs

The first prototype of the HERCULE display was tested by eight subjects. The subjects were specifically chosen as being computer-illiterate, since it was felt that the use of computing science students for this type of experiment would produce an unrealistic result. A very simple application was used, which allowed users to carry out simple banking transactions on various accounts. The choice of a banking application was made because of the familiarity of the general populace with this type of computer application and because it consequently did not intimidate the subjects. The application interface was fairly simple and allowed the user to click on buttons to make choices of the type of banking transaction — opening or closing an account, depositing or withdrawing funds. Inputs were provided by means of text fields. The following results were obtained:

1. Various errors were deliberately generated throughout the experiment and the users were observed dealing with the errors. Users also spontaneously made unforced errors which enriched the experiment considerably. They did handle the errors better when the HERCULE display was visible and seemed more confident and relaxed when they understood the problem.

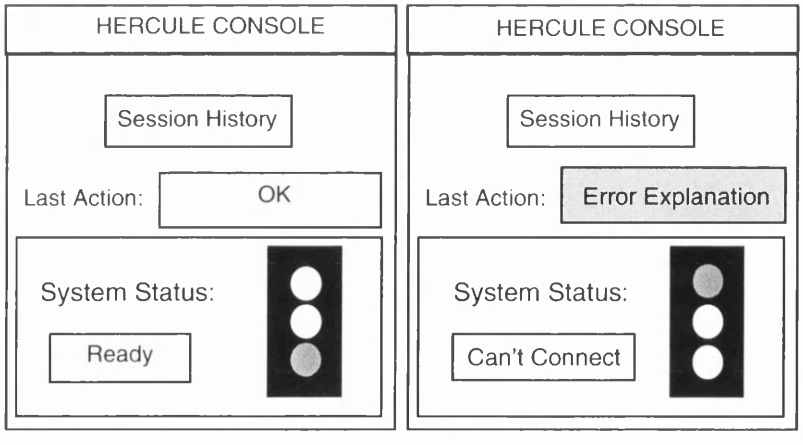

Figure 8.1: The Initial Display

2. Users were asked directly whether they felt that the display had been helpful. In retrospect this was unwise, since they almost all felt obliged to be complimentary about it. When asked to rate their performance with and without the HERCULE display, the majority rated their performance as being better *with* the display. This too, was suspect, due to the previously mentioned *Hawthorne Effect.*

3. The initial prototype feedback display, shown in Figure 8.1, required the user to click on a button to get an explanation of the error and this caused some irritation in at least one of the subjects, who wanted the explanation offered without having to go looking for it.

   The first prototype provided the user with archival feedback in the form of a table, as shown in Figure 8.2, with clickable buttons beneath the "Action" and "Effect" headers to give users more information about their actions, and the corresponding system response. The effort required by the user to get at the needed information caused the same irritation as mentioned above. The table is also clearly not scalable and was not a good solution.

SESSION HISTORY			
TIME	ACTION	EFFECT	SUCCESS/ FAILURE
3:01	Find Book	More...	SUCCESS
3:05	Search	More...	SUCCESS
3:15	Order	More...	SUCCESS
3:16	Submit	More...	FAILURE

Figure 8.2: The Initial Session History Display

The conclusions which can be drawn from the users' reactions to this display underline the findings described in Chapter 4. Users simply do not want to spend time looking for answers to questions. They want the information to be directly available — supporting an increasingly likely "situated action" mode of operation. This experience led to the format of the present display (Figure 8.3), which gives an explanation of the most recent activity spontaneously without any effort on the part of the user.

4. It was also noted that users often did not detect errors, even though they were being reported by the application in the form of error messages. This led to the inclusion of the optional beep feature into the HERCULE display — which alerts users to the occurrence of an error. It is optional because it might not be suitable in a noisy environment to use a beep, or the user's aversion to a beeping noise may negate the positive effects of the beep.

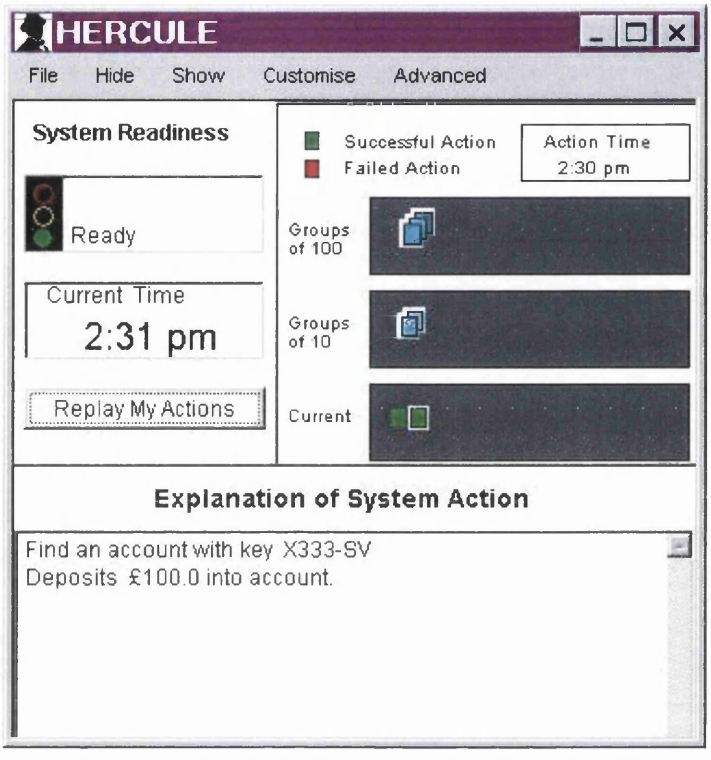

Figure 8.3: The Revised Session History Display

The experimental use of HERCULE was obviously valuable in discovering problems with the display, but there was still a feeling of unease with respect to the fact that the simplicity of the application made the HERCULE display less useful than it could be. This led to the decision *not* to repeat the experiment with the latest HERCULE display, but rather to rely on a *functional* evaluation of the features offered. This will be addressed in the following sections.

8.2.1.1 Feedback

Immediate Feedback

The immediate feedback will be evaluated according to the features listed in Section 4.5.2.

1. *System state indicator* — HERCULE provides a continuous feedback mechanism in the form of traffic lights. The lights are green when the system is idle and waiting to be used. The lights are orange when the system is busy servicing a request and red when the system has broken down and cannot be used. The traffic lights were used because 2% of the population is colour-blind and it is not sufficient to have an indicator which is either red, green or orange. The traffic light structure is universally recognised and even colour-blind people will have no difficulty interpreting it. Traffic lights are also

used throughout the world, and will be readily assimilated by all cultures and across language barriers. Another helpful indicator of system state is the time display.

2. *Explanations* — The HERCULE provides a spontaneous explanation of the system's actions as a result of the user's actions. This is provided in the format preferred by the user, so that the feedback provided to programmers will be very different from that provided to the end-user or system support person. This feedback can also be tailored to suit users' particular language of choice or mode of communication.

3. *Making visible what is often invisible* — The *effects* of actions are made visible, by means of the above-mentioned explanations. In the absence of such feedback, the user can only guess at what the system did as a result of their inputs.

Archival Feedback

1. *Mental aids* — provided by the context building facility, which performs an action replay and enables reconstruction of the mental context surrounding a specific task.

2. *Inter-referential links* — provided by the overall history display. HERCULE presents an overall display of the session history which graphically depicts Episodes — each being a direct link between actions taken and the success or failure of system activity precipitated by those actions. This "overview and zoom" technique allows the user to get information about previous actions and their effects.

8.2.1.2 Quirks

Chapter 3 described what were called *quirks*, those things which interfere with "normal" execution of a task. The characteristics of the three sub-groups of quirks were described and their effects on the user explored. The following sections explain how HERCULE can alleviate the negative effects of quirks.

Error

Section 3.5 described error in some detail. While HERCULE cannot prevent errors, it can ease the detection of, understanding of, and recovery from, errors:

- *Error detection* — HERCULE could go some way towards reducing the time elapsed before the error is detected, by providing feedback about actions taken. Information about inputs given and results obtained from the server are all recorded and can be accessed by the user. There is a flaw in this though, because the user who has made this type of error has no reason to confirm an action unless some feedback mechanism, or some difference in the state of the application, makes evident the fact that something has gone wrong. Enhancing feedback by displaying a window containing a confirmatory message after the action has completed would help reduce

the occurrence of this state, but HERCULE's non-invasiveness property prevents this course of action.

Since users do not always see error messages, HERCULE can assist by providing visual and audio feedback about the success or failure of an action. Non-detection of this type of error state is not as damaging as it is for undetected errors, since the data store has not been affected.

The Episode display colour indicates the presence of an error by displaying red and by beeping, unless the user chooses to deactivate the beep feature. The use of colour was discovered not to be sufficient in HERCULE's first end-user evaluation, and this influenced the decision to incorporate an optional beep-upon-error facility. It is possible to flash an error message window, but there is a need to proceed carefully since the user should not be annoyed any more than they will be already by the presence of an error.

- *Error understanding* — There are two aspects involved in making the problem clear to the user:

  1. the first is a reminder of what the person did, and
  2. the second is an explanation of what the system did as a result of the action.

  HERCULE provides a small rectangle, as part of the session history, representing this link, while the information about either the user actions or the system actions can be obtained with ease.

- *Error recovery* — Once users understand the nature of the error, and their part in causing it, the next step is to assist them in recovering from the error. They should be able to work out what steps to take in getting the application to the state they intended. HERCULE reduces cognitive under-specification [Rea90] according to the guidelines given in [RPMB96]:

  1. *Make the action perceptible* — HERCULE links the action to the effect. The users can link their inputs to the system's actions and this should help them understand why the error occurred.
  2. *Display message at high level* — explanations of error are given in terms of the user's intentions.
  3. *Provide an activity log* — provided by the archival feedback facility.
  4. *Allow comparisons* — not supported.
  5. *Make action result available to user evaluation* — provided by means of the archival facility.
  6. *Provide result explanations* — the results of the method invocations are explained.

In general terms, HERCULE should assist the user to build up an internal model of how the system works, enabling them to move more quickly toward the skill-based level of performance. Table 3.1 shows that if the time taken to resolve an error can be reduced, it will directly reduce the negative emotions experienced by the user. HERCULE seeks to reduce this time by explaining system actions so that the user does not have to puzzle about things for too long and get annoyed.

Interruptions

The biggest problem caused by interruptions, as explained in Section 3.6, is the re-establishment of the context after an interruption. The user will often not go back to the original task but resume another task altogether — often with disastrous results[1].

Chapter 4 introduced Suchman's [Suc87] situated action theory, which presents the case that users tend to respond to their current circumstances rather than following a rigid plan. Once the user has lost context, it often does not help to look at the last window displayed by the application. When using a browser it is a simple task to backtrack over the browsing path to check previous states. If this were not available, they may remember what they were doing, but will often have difficulty doing so.

Users need to be supported in linearising the activities of their regular working day and in dealing with unexpected interruptions. To help with this, a software application should facilitate the rebuilding of the lost *context* so that the user remembers the circumstances which existed, with respect to the application, before the interruption.

Since the user's interaction with modern computer systems is essentially based on recognition, rather than recall, and is intensely visual, it would be less than optimal to try to describe a set of user or system actions in a textual format. Therefore, to assist the user in rebuilding the mental context after an interruption HERCULE provides an *action replay* of the user's interaction with the application — up to the point of the interruption. Since the original circumstances were established based on the recognition of certain windows on the screen, the reconstruction of this state is facilitated, and eased, by visual means as well.

Breakdowns

Section 3.4 discussed breakdowns and identified four possible breakdown locations:

1. *The end-user computer itself* — the crash of the entire computer will mean that HERCULE will cease execution too. Only if the session history is made persistent can it be useful in such a case.

2. *An application on the end-user computer* — errors generated as a result of communication between the a component-based application and the rest of the component-based

[1] It is a rare person who has never allowed the bath to overflow or the supper to burn!

system will always be signaled by exceptions. HERCULE will provide a user-friendly explanation of the error, rather than confronting the user with an exception printout.

3. *The network* — this could be signaled by a lack of response. HERCULE times responses, tries to find out what is wrong and provides an explanation.

4. *The application server and/or data store* — this will present as either an unlimited delay, which will be handled as network errors are handled — or by an exception, which will be handled as an application error.

HERCULE provides continuous feedback to keep the user informed about the state of the entire component-based application. This will help the user to identify errors outside the scope of their own computer. The current **Task Manager** facility offered by Windows operating systems offers the useful facility for being able to ascertain that applications on one's own machine are not responding. HERCULE attempts to provide this facility for applications outside the user's machine. There is, as yet, no way for HERCULE to detect application, hardware or software faults.

8.2.2 Component-Based System Development and Maintenance

Section 2.5 pointed out that component-based development is a relatively new field and that it is logical to expect the development process to change and mature as component-based development becomes the order of the day. The programmer's task is thus not clearly defined at present. HERCULE seeks to support programmers in their efforts to produce a good product in what is becoming an increasingly complex environment. The 21st century programmer has a much more complex task and the concepts which must be mastered are a world removed from those of the programmer of the last two decades. The following section will explain how HERCULE assists the programmer in providing feedback, while Section 8.2.2.2 summarises a programmer's experience with HERCULE.

8.2.2.1 Programmer Needs

Section 5.1.1 concluded that programmers are not often trained to provide good user interfaces and that for a number of reasons the feedback provision by programmers seems doomed to be inadequate. The HERCULE framework acknowledges this and seeks to make the task easier for the programmer. Programmers must participate in tailoring messages for the end-user, by means of one of the HERCULE tools and that is their only contribution.

In return for this small investment, programmers get help in debugging programs, since the framework times responses and displays information about method invocations and server replies. They also save time in generating user-friendly error messages within their programs, since the framework will do this for them.

The focus in this research has been the simplification of the programmer's task, since it is my understanding that they have more than enough to do in programming the core system functionality.

8.2.2.2 Programmer Experience with HERCULE

There are two ways to evaluate this type of software, from the programmer's perspective:

1. *objective* evaluation by a number of programmers. The author's approaches to industry were met with polite refusals. It could have been that companies were afraid to demonstrate their use of technology to an outsider, which is understandable in the cut-throat software industry of today. The ever prevalent deadlines in this industry which considers 3 month plans to be "long-term" planning, make the possibility of testing tools in a real world environment rather unlikely.

   Faced with this brick wall, the other option, that of using student programmers within the University to test the software, was attempted, with little success. It proved to be extremely difficult to find enough programmers with the required expertise within the University. An attempt was made to interest the MSc class, but, when faced with the steep learning curve required to develop an application using EJBs the students were unwilling to participate. There is also the difficulty of persuading students to put a lot of effort into a project for which they are not earning credits.

2. *subjective* evaluation by one or two programmers. While not the perfect solution, this proved to be the only workable evaluation method and was thus the approach followed.

The following discussion addresses the experience of HERCULE as obtained via an interview with a programmer who used it to develop a thin client for a CBS. The general feedback can be summarised as follows:

- He enjoyed the fact that he did not have to do anything special for HERCULE to work. He could program in his own style, using his own mechanisms, without worrying about HERCULE. This meant that the "ease of insertion" criteria scored highly. Since no extra effort was required to facilitate HERCULE's functioning, there was total programmer "ease of use".

- The customisation was easy to use, as he found that it took very little time — scoring high on end-user "ease of use".

- HERCULE did not crash. (This robustness was particularly encouraging)

- He felt that the HERCULE feedback component which displayed details of method invocations raised his productivity since he did not have to track his own application and print the details out himself.

He suggested some changes which were incorporated in the final version:

1. the need for more information about exceptions was expressed. At that stage HERCULE did not include any information about the method invocation causing the exception, in the exception explanation.

2. the time taken by the method invocation, if included in the programmer feedback component display, was seen to be very helpful.

3. a change to the highlight mechanism was suggested. The HERCULE display currently uses the highlighting technique suggested by the programmer. The previous display used a triangle to indicate the current Episode and he quite rightly pointed out the inconsistency this caused.

8.2.3 Performance Impact

It is very important that the presence of HERCULE should not affect application performance unacceptably. Since HERCULE inserts proxies between the application and the user interface, and between the application and the rest of the CBS, we can expect any performance degradation to take place:

1. when the user interface proxy is being loaded, since an extra level of indirection is being introduced;

2. when the initial connection with the server is being forged, since this is where the server proxy will be introduced;

3. whenever a new window is being constructed; and

4. when global methods are invoked on distributed components. Two types of methods need to be considered independently, methods which will require action by the component container and methods on the component itself. The former naturally take longer than the latter to process.

A preliminary study of performance differences was undertaken, by running the example application twenty times both with and without HERCULE. The client computer used was a Pentium 166, not exactly the leading edge of computing technology, and the figures should therefore be seen as a "worst-case scenario". Where effects were observed, the results are shown, in seconds taken for each activity, in Table 8.1[2].

It is clear that the user *will* have to pay a penalty for using HERCULE. It would be unreasonable to expect otherwise. The entire insurance industry is based on the "present pain, future gain" principle. Shneiderman [Shn98] cites research which shows that modest

[2]There was no discernable effect when new windows were constructed, with only the time taken for the initial window being affected.

Action	Without Proxies	With Proxies
Display of Initial Application Window	1.44	2.45
Initial Contact with Server	5.92	8.73
Container Method Invocation	1.40	1.56
Component Method Invocation	0.25	0.54

Table 8.1: Time taken for core activities

variability in response times is deemed by users to be acceptable. If users see the benefits of using HERCULE, they will hopefully be prepared to pay the small penalty of slightly longer response times, for the future gain of having informative and extensive feedback available.

8.3 Conclusion

The evaluation of HERCULE is by no means completed. I have cited it as a topic for future research in Chapter 9. The evaluation of this type of specialised software tool will be no small task, since the world of component-based systems is a relatively young field, and the skills required to operate as a programmer in this area are not yet commonly found.

However, such evaluation as has proved possible has indicated that HERCULE does convey some benefits to end-users and application programmers alike. End-users experience improved feedback while programmers find it easy to use, and find that it assists in application development by automatically tracking the application. Performance is somewhat affected, but not in such a way that the end-users will be significantly disadvantaged. I look forward to pursuing this line of research in the future.

I am sorry that I have had to leave so many problems unsolved. I always have to make this apology, but the world really is rather puzzling and I cannot help it.

Bertrand Russell.
"The Philosophy of Logical Atomism," Lecture V

chapter 9

Conclusion

9.1 Reiteration of Thesis Statement

I submit that feedback can be enhanced in a distributed component-based system by executing the application within a generic feedback enhancing framework. I further submit this supports the user: firstly in understanding the application, secondly in recovering from errors, and thirdly in rebuilding mental context after interruptions. The framework standardises feedback provision, simplifies application code, allows continuous post-implementation refinement of explanatory messages and promotes reuse.

9.2 Summary of Research

This dissertation started off by drawing a comparison between human-to-human conversation and human-to-computer interaction, and concluded that the ability of computer applications to generate a shared context with the user needed to be enhanced. This conclusion was based on personal experience, with many professional people who happened to be computer-illiterate, vast amounts of anecdotal evidence and the prevalence of web-sites and newspaper columns explaining the behaviour of computer applications and interpreting error messages for the benefit of perplexed end-users.

The introduction stated the author's intent to explore the enrichment of the traditional feedback provided by applications to enhance the interpretability of applications and provide an explanatory bridge between the application programmer and the end-user. It is abundantly clear from the applications in use today that the provision of feedback is sadly neglected. Current techniques clearly needed to be re-examined and a new approach found.

Component-based systems are being used increasingly in all types of systems and in these systems the possibility that the user will receive adequate feedback is even smaller than usual. The nature of component-based systems was explored, with Chapter 2 providing an overview of the current state of this technology.

While users might have problems in using these systems when things are proceeding as planned, they could have even more problems if something interferes either with their concentration or execution of the task. The nature of the various events which could interfere with straightforward execution were studied and a classification of *quirks*, which encompassed all such events, was derived. Quirks can be either breakdowns, human errors or interruptions. The characteristics of each type of quirk were explored in depth in Chapter 3.

The literature with respect to feedback was studied and the conclusion drawn that feedback could be either *immediate* or *archival*. Each type meets different needs — immediate feedback assisting the user in understanding the rules of discourse, and archival feedback making the application state visible. This boundary is not absolutely rigid, with immediate and archival feedback fulfilling the other's function as well. While feedback can be very useful when everything proceeds according to plan, it becomes even more essential when something goes wrong, or interrupts the user's interaction with the application. The use of feedback to alleviate these negative effects was investigated. The findings with respect to feedback were given in Chapter 4.

Having thereby motivated the need for an additional and augmentary user-programmer communication mechanism, an approach was developed which provides the end-user with a runtime feedback assistant, named HERCULE, which can also be used as a software-development tool to ease the programmer's task. This approach combined established techniques of application tracking, separation of concerns and visualisation to provide the end-user with a visualisation of application activity (described in Chapter 5). The design and implementation of this tool was discussed in Chapters 6 and 7. This unique form of feedback — application activity visualisation — augments the feedback provided by a component-based application, so that the end-user is assisted in understanding and using these applications, as discussed in Chapter 8. This is expected to satisfy many of the user's feedback needs, as has been suggested by initial usability tests. The approach applied, and mechanism developed, during the course of this research is applicable to a wide range of end-user applications. Thus, although this dissertation has concentrated on the provision of this framework in the context of component-based systems, its scope is far wider, and can be applied as such.

To conclude, this dissertation has shown that it *is* feasible to provide feedback, using a combination of separation of concerns and application tracking, as a visualisation of the application activity and has developed initial evidence to strongly suggest this will often be beneficial to both application developers and end-users.

9.3 Thesis Contribution

The contributions of this dissertation can be enumerated as follows:

1. A summary of the large and volatile field of component-based systems, a field which does not easily lend itself to scientific analysis. This is due to changing names, different meanings attributed to the same names, prevalence of books which are designed for managers rather than engineers and scientists, and marketing jargon. Chapter 2 describes how components have evolved, explains issues with respect to component-based development and gives a brief overview of the prominent component models in use today.

2. A classification of quirks, those diverse events which interfere with our everyday execution of tasks. Each type of quirk — error, interruptions, and breakdowns — was analysed and classifications derived. Findings with respect to quirks were given in Chapter 3.

3. Motivation for the extension of the traditional concept of feedback to include archival feedback as well as immediate feedback. A case was also made for the due consideration of the use of graphical feedback rather than solely textual descriptions. The need for customisability of feedback to meet the needs of different types of users or users functioning in different roles was also addressed. Feedback was discussed in Chapter 4.

4. A review and organisation of several normally unrelated areas of research — separation of concerns, application tracking and visualisation — into one framework for future reference, in Chapter 5.

5. Motivation for treating feedback as a separate concern and for implementing this separation by means of application tracking, also in Chapter 5.

6. Development of a model of application activity, namely Episodes, to be portrayed as being representative of the activity of the application, and motivation for providing feedback graphically.

7. A prototype implementation of the proposed framework, which tests the viability of the proposed scheme and provides a visualisation of the activity of the application, described in Chapters 6 and 7.

8. A design pattern, namely the "Minimal Impact Proxy" pattern, was developed to be re-used in ensuring that proxies do not impair application performance more than they should. This is discussed in Section 6.2.1

9.4 Future Research

The author echoes the sentiments of Bertrand Russell in feeling that many issues have been left unsolved. It is some consolation that this is the nature of research and that it does give one many opportunities for further work. The following opportunities for future research have been identified:

1. *Enhanced query facilities for the display.* The display as it stands does not offer many opportunities for either grouping of similar Episodes, or searching for a specific Episode, or characterising Episodes as one of a certain type. Since the display is essentially a type of visualisation these facilities will have to be provided if HERCULE is going to be a meaningful tool.

2. *Link to knowledge base to explain errors.* Dellarocas [Del98] has developed a scheme whereby a knowledge base is established which builds up a collection of explanations of errors. This would take some of the effort out of defining the reasons for exceptions, and assist the programmer, since the same explanation could be used throughout the application.

3. *Incorporating fault tolerance.* Huang and Kintala [HK93] have worked on an add-on fault tolerance mechanism which is widely used within their organisation (Bell Labs). This could conceivably be harnessed by HERCULE. For example, HERCULE could be used to detect the failure of a particular server and notify a specified person so that the problem can be resolved in as little time as possible.

4. *Full evaluation of HERCULE*, which is necessary to confirm conclusively the many benefits of HERCULE both to the end-user and programmer. It is not absolutely clear how this evaluation should be done, since this tool is a new concept in software development. It would be interesting to work with researchers in the evaluation field to arrive at a comprehensive evaluation method and thereby be able to assess HERCULE comprehensively.

5. It would obviously be very helpful for HERCULE to *keep a permanent record of session activity.* This would be very helpful for auditing, security purposes etc.

6. It has been suggested that it would be helpful if HERCULE *could run on a remote machine*, so that a system support person could monitor the performance of a particular application on a particular machine from their own office.

7. The mechanism used to insert Java proxies opens up a host of questions about the *safety and security* of Java applications, which I look forward to investigating in more depth in the future.

8. The HERCULE concept could easily be adapted to function as a *monitor of user interface usability.* It could be used to check which parts of the dialogue were used regularly, which were ignored, and which were seldom invoked. Instead of supplying the user with an activity visualisation, HERCULE could be tailored to write such information to files so that it could be analysed by the usability engineer.

— The End —

part VI

Appendices and Bibliography

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started.
T S Eliot. 1944

Appendix A

Glossary

ActiveX — COM based visual desktop components integrated into applications.

ANSA — *Advanced Networked Systems Architecture.*

API — *Application Programmer Interface.*

CBD — *Component Based Development.* The process of designing and developing a system using pre-built components.

CBS — *Component Based System.* The system built using components.

COM — *Component Object Model.* Microsoft's component model.

CORBA — *Common Object Request Broker Architecture.* The component model specification delivered by the OMG.

CVC — *Component Vendors Consortium.* An organisation which seeks to standardise technical support and documentation of components.

DBMS — *Database Management System.*

DCOM — *Distributed COM.* An extension of COM which allows components to reside on different machines.

DTC — *Distributed Transaction Coordinator.* Ensures consistency in the face of multi-database transactions.

EJB — *Enterprise Java Bean.* Sun's component model.

GUI — *Graphical User Interface.*

HCI — *Human Computer Interaction.*

HTML - *Hypertext Markup Language.* Formatting language used by Web Documents.

IDL — *Interface Definition Language.* Defines component interfaces.

IIOP — *Internet InterORB Protocol.* Protocol for different ORBs to interoperate.

IPC — *Inter Process Communication.* Protocol for applications to communicate by means of sockets.

IR — *Interface Repository.* Repository of component interfaces used by CORBA.

JNDI — *Java Naming and Directory Interface.* Naming service for the EJB standard.

JTS — *Java Transaction Service.* Transaction monitor for Sun's EJB.

JVM — *Java Virtual Machine.* The portable virtual machine for applications written in the Java language.

MTS — *Microsoft Transaction Server.* Component-oriented middleware for COM.

OCX — 32 bit version of VBX.

OLE — *Object Linking and Embedding.* First Microsoft components.

OMG — *Object Management Group.* Authors of the CORBA standard.

ORB — *Object Request Broker.* CORBA protocol for interacting with remote objects.

OTM — *Object Transaction Monitor.* Another terminology for component-oriented middleware.

OTS — *Object Transaction Service.* CORBA's specification for distributed transactions.

PC — *Personal Computer.*

RMI — *Remote Method Invocation.* Allows method invocations of remote objects in the same way as is done locally.

RPC — *Remote Procedure Call.* Procedure call protocol implemented for client-server architectures.

TPM — *Transaction Processing Monitor.*

URL — *Universal Resource Locator.*

VBX — Visual Basic Controls (application-internal components).

XML — *Extensible Markup Language.*

Appendix B

Minimal Impact Proxy Design Pattern Code

Proxy Code Fragment

The setting up of the link to the `ReporterQueue` is demonstrated in the following code:

```
ReporterQueue queue;

public Proxy() {
   queue = new ReporterQueue();
   queueThread = new Thread(queue);
   queueThread.start();
} // constructor
```

ReporterQueue Code Fragment

```
public class ReporterQueue extends Thread {

// THE TARGET FOR ALL OUR REPORTS
```

```
Reporter reporterToBeNotified;
// other variables ....

public ReporterQueue(){
   // start up the communicator object
   reporterToBeNotified = new Reporter(this);
   Thread reporterThread = new Thread(reporterToBeNotified);
   reporterThread.start();

   // wait till the thread is alive and running
   while (!reporterThread.isAlive()) ;
} // constructor

public void run() {} // required to implement Runnable

public synchronized void addItem(ReporterQueueItem newItem){
//   ADD A NEW REPORTER QUEUE ITEM TO THE QUEUE
// ....
// tell the waiting thread something is in the queue
reporterToBeNotified.wakeUp();
} // addItem
} // ReporterQueue
```

Reporter Code Fragment

```
public  class Reporter  extends Thread {

static boolean verbose = Boolean.getBoolean("verbose");
ReporterQueue queue;

static boolean reportEvents=true; // report till false
static java.net.Socket socket;
static java.io.OutputStream outStream;
static java.io.ObjectOutputStream objectOut;

public Reporter(ReporterQueue rq) {
   queue = rq;
   setOutputStream();
} // constructor

public synchronized void wakeUp() {notify();}

public void run() {
```

```
try {
   while (true) {
      if (queue.elements() == 0)
      synchronized (this) {
         if (reportEvents) wait(1000); // wait for something
         else wait(); // ERROR CONDITION - go to sleep forever
      } // nothing on the queue
      if (queue.elements() > 0) {
         // GET THE FIRST REPORT OFF THE QUEUE
         ReporterQueueItem item = queue.getHead();
         write(item.getReport());
      } // elements on the queue
   } // while
} // try
catch (Exception ee) {
   if (verbose) {
      System.out.println("EXCEPTION - Reporter");
      ee.printStackTrace();
   }
} // catch
} // run()

public void setOutputStream(){
// THIS IS ONLY DONE ONCE, TO SET UP COMMUNICATIONS WITH
// THE MONITORING APPLICATION
// ....
} // setOutputStream()

public void write(Report message) {
if (!reportEvents) return; // an error occurred, do nothing
try { objectOut.writeObject(message);}
catch (Exception ee) {
   if(verbose) {
     System.out.println("Problem communicating");
     ee.printStackTrace(System.out);
   }
} // catch
} // write
} // Reporter
```

Bibliography

[ABvdSB94] Mehmet Aksit, Jan Bosch, William van der Sterren, and Lodewijk Bergmans. Real-Time Specification Inheritance Anomalies and Real-Time Filters. In *[TP94]*, pages 386–407, 1994.

[ACT94] P V Argade, D K Charles, and C Taylor. A Technique for Monitoring Run-Time Dynamics of an Operating System and a Microprocessor Executing User Applications. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA USA, Oct 5-7 1994. ACM.

[All99] P Allen. Doing Business with Component-Based Development. *Application Development Advisor*, 3(1):36–44, October 1999.

[AM82] C M Allwood and H Montgomery. Detection of errors in statistical problem solving. *Scandinavian Journal of Psychology*, 23:131–139, 1982.

[AM95] M. P. Atkinson and R. Morrison. Orthogonally Persistent Object Systems. *VLDB Journal*, 4(3):319–401, 1995.

[And83] J R Anderson. *The Architecture of Cognition.* Harvard University Press, Cambridge, Massachusetts, 1983.

[And90] P B Andersen. *A Theory of Computer Semiotics.* Cambridge Series on Human-Computer Interaction. Cambridge University Press, Cambridge, 1990.

[ANS89] ANSA. An Engineers Introduction to the Architecture. Architecture Projects Management Limited. Cambridge, November 1989. Release TR.03.02.

[Aoy98] Mikio Aoyama. New Age of Software Development: How Component-Based Software Engineering Changes the Way of Software Development. In *[ics98]*, 1998.

[App87] *Human Interface Guildelines: The Apple Desktop Interface.* Addison-Wesley, Reading, Massachusetts, 1987. Apple Computer Inc.

[Asp98] http://www.parc.xerox.com/spl/projects/aop/aspectj. AspectJ Web Page, 1998.

[AST99] M Aleksy, M Schader, and C Tapper. Interoperability and Interchangeability of Middleware components in a Three-Tier CORBA-Environment-State of the Art. In *3rd International Enterprise Distributed Object Computing Conference. EDOC'99*, pages 204–213, University of Mannheim, Germany, September 27-30 1999. IEEE.

[Bac97] J Bacon. *Concurrent systems: operating systems, database and distributed systems: an integrated approach.* Addison Wesley, Harlow, second edition, 1997.

[Bae98] T Baer. The Culture of Components. Web Document, September 1998. www.adtmag.com/pub/sept98/fe902.htm.

[Ban89] L Bannon. From Cognitive Science to Cooperative Design. In N O Finneman, editor, *Theories and Technologies of the Knowledge Society*, pages 33–59. 1989.

[Bas99] P Bassett. Two Flavors of Component Architectures. *Component Strategies*, 1(11):70–72, May 1999.

[BDPS94] Marc Brown, John Domingue, Blaine Price, and John Stasko. Software Visualization. *ACM SIGCHI Bulletin*, 26(4):32–35, 1994.

[Beg99] James Begole. Personal communication, March 1999.

[BF88] L Blackshaw and B Fishhoff. Decision making in online searching. *Journal of the American Society for Information Science*, 39:369–389, 1988.

[BF95] J R Brooks and P Frederick. *The Mythical Man-Month.* Addison-Wesley, 1995.

[BF97] D J Berg and J S Fritzinger. *Advanced Techniques for Java Developers.* John Wiley and Sons, 1997.

[BG93] Thomas Berlage and Andreas Genau. A Framework for Shared Applications with a Replicated Architecture. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, CSCW and Distributed Applications, pages 249–257, 1993.

[BH93] Susan E. Brennan and Eric A. Hulteen. Interaction and feedback in a spoken language system. In *AAAI Technical Report FS-93-05*, pages 1–6, 1993.

[BK98] A N Burton and P H J Kelly. Workload Characterization and Using Lightweight System Call tracing and re-execution. In *IEEE International Performance Computing and Communications Conference. IPCCC '98*, Phoenix/Tempe, Arizona, USA, February 16-18 1998. IEEE.

[BK99] A N Burton and P H J Kelly. Tracing and Reexecuting Operating System Calls for Reproducible Performance Experiments. *Computers and Electrical Engineering: An International Journal*, May 1999.

[BL94] T Ball and J R Larus. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, 1994.

[BMP+99] Peter J. Barclay, Jo McKirdy, Norman W. Paton, Philip D. Gray, Jessie Kennedy, Richard Cooper, Carole A. Goble, Adrian West, and Michael Smyth. Teallach: A model-based user interface development environment for object databases. In Norman W. Paton and Tony Griffiths, editors, *Proc. User Interfaces to Data Intensive Systems (UIDIS99)*, pages 86–96, Edinburgh, Scotland, 5–6 September 1999. IEEE Computer Society Publishers.

[BN84] A D Birrell and B J Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[Bor91] N Borenstein. *Programming as if People Mattered.* Princeton Univeristy Press, Princeton, New Jersey, 1991.

[Bro99] A W Brown. Moving from components to CBD. *Component Strategies*, 1(10):23–28, April 1999.

[BW97] M Büchi and W Weck. A Plea for Grey-Box Components. In *[LS97]*, 1997.

[BW98] Alan W Brown and Kurt C Wallnau. The Current State of CBSE. *IEEE Software*, 15(5):37–46, September/October 1998.

[BZPF93] F C Brodbeck, D Zapf, J Prümper, and M Frese. Error handling in office work with computers: A field study. *Journal of Occupational and Organizational Psychology*, 66:303–317, 1993.

[Car87] J M Carroll, editor. *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction.* MIT Press, Cambridge, MA, 1987.

[CCH00] Edward B Cutrell, Mary Czerwinski, and Eric Horvitz. Effects of Instant Messaging Interruptions on Computing Tasks. In Barbara Hayes-Roth and Richard Korf, editors, *Proceedings of CHI'2000*, pages 814–819, The Hague, Netherlands, 1-6 April 2000.

[CCS91] M Czerwinski, S Chrisman, and B Schumacher. The Effects of Warnings and Display Similarities on Interruption in Multitasking Environments. *SIHCHI Bulletin*, 23(4):38–39, October 1991.

[CE89] F R Campagnoni and K Ehrlich. Retrieval using a hypertext-based help system. *ACM Transactions on Information Systems*, 7:271–291, 1989.

[Cha96] D Chappell. *Understanding ActiveX and OLE.* Strategic Technology Series. Microsoft Press, Redmond, Washington, 1996.

[Cha97] D Chappell. The Next Wave. Component Software Enters the Mainstream. Web Document, April 1997. Chappell and Associates. www.chappellassoc.com.

[Cha98] D Chappell. MTS versus EJB. *Component Strategies*, 1(5):14–17, November 1998.

[Cha99a] Matthew Chalmers. Information visualization tutorial. In *25th International Conference on Very Large Data Bases VLDB'99*, Edinburgh, Scotland, 7th - 10th September 1999. Morgan Kaufmann.

[Cha99b] D Chappell. Application servers: COM-Based vs. Java-Based. *Component Strategies*, 1(9), March 1999. www.chappellassoc.com/art1cs.htm.

[Cha99c] D Chappell. Taking Stock of Component Technology. *Component Strategies*, 1(12):16–17, June 1999.

[Che99] Chaomei Chen. *Information Visualisation and Virtual Environments.* Springer, Singapore, 1999.

[CK98] John V. Carlis and Joseph A. Konstan. Interactive Visualization of Serial Periodic Data. In *Proceedings of the ACM Symposium on User Interface Software and Technology. UIST'98*, Visualization, pages 29–38, San Francisco, CA USA, November 1 - 4 1998.

[Cla97] William J Clancey. *Situated Cognition. On Human Knowledge and Computer Representations.* Cambridge University Press, Cambridge, UK, 1997.

[CM93] Giuseppe Carenini and Johanna D. Moore. Generating Explanations in Context. In *Proceedings of the 1993 International Workshop on Intelligent User Interfaces*, Session 6: User Support, pages 175–182, 1993.

[CMH92] Elliot J. Chikofsky, David A. Martin, and Chang Hugh. Assessing the State of Tools Assessment. *IEEE Software*, 9(3):18–21, May 1992.

[CMN83] S Card, T Moran, and A Newell. *Applied Information-Processing Psychology.* Erlbaum Associates, Hillsdale, NJ, 1983.

[Cor91] John R Corbin. *The Art of Distributed Applications.* Springer Verlag, New York, 1991.

[Cot98] B Cottman. Componentware: Component software for the enterprise. http://www.i-kinetics.com/wp/cwvision/CWVision.html, 1998. (27/11/98) I-Kinetics Web Site.

[Cox90] B Cox. There is a silver bullet: The birth of interchangeable, reusable software components will bring software into the information age. *Byte*, October 1990.

[CR87] J M Carroll and M B Rosson. The Paradox of the Active User. In *[Car87]*, chapter 5, pages 80–111. MIT Press, 1987.

[CRB98] M Chalmers, K Rodden, and D Brodbeck. The Order of Things: Activity-Centred Information Access. In *Proceedings of the 7th International Conference on the World Wide Web*, pages 359–367, Brisbane, Australia, Oct 5-7 1998.

[CRM91] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The information visualizer, an information workspace. In *Proc. ACM Conf. Human Factors in Computing Systems, CHI*, pages 181–188. ACM, April 1991.

[CS87] H H Clark and E F Schaefer. Collaborating on contributions to conversations. *Language and Cognitive Processes*, 2:1–23, 1987.

[CWS95] H C Chan, K K Wei, and K L Siau. The effect of a database feedback system on user performance. *Behaviour and Information Technology*, 14(3):152–62, 1995.

[Cyp86] Allen Cypher. The structure of users' activities. In D A Norman and S W Draper, editors, *[ND86]*, chapter 12, pages 243–264. Lawrence Erlbaum Associates, Publishers, Hilldale, New Jersey, 1986.

[Das92] Marcelo Dascal. On the Pragmatic Structure of Conversation. In Herman Parret and Jef Verschueren, editors, *(On) Searle on Conversation*, pages 35–56. John Benjamins Publishing Company, Amsterdam, 1992.

[dB98] Edward de Bono. *Simplicity.* Penguin, London, 1998.

[De 98] J. De Oliveira Guimaraes. Reflection for Statically Typed Languages. *Lecture Notes in Computer Science*, 1445:440–461, 1998.

[Del98] C Dellarocas. Toward Exception Handling Infrastructures for Component-Based Systems. In *[WCB88]*, 1998.

[DFAB93] A Dix, J Findlay, G Abowd, and R Beale. *Human-Computer Interaction.* Prentice Hall, 1993.

[DH95] Nick Drew and Bob Hendley. Visualising Complex Interacting Systems. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, volume 2 of *Short Papers: Information Visualization*, pages 204–205, 1995.

[Dix91] A J Dix. Closing the loop: modelling action, perception and information. In M. F. Costabile T. Catarci, S. Levialdi, and G. Santucci, editors, *AVI'96 - Advanced Visual Interfaces*, pages 20–28. ACM Press, 1991.

[DJA93] Nils Dahlback, Arne Jonsson, and Lars Ahrenberg. Wizard of Oz Studies – Why and How. In *Proceedings of the 1993 International Workshop on Intelligent User Interfaces*, Session 7: Design & Evaluation, pages 193–200, 1993.

[Dol98] M Dolgicer. Distributed Object Middleware & the Internet. *Component Strategies*, 1(5):23–32, November 1998.

[Dol99] M Dolgicer. Building a Middleware Platform. *Component Strategies*, 1(9):40–44,57, March 1999.

[DPM92] T M Duffy, J E Palmer, and B Mehlenbacher. *Online Help. Design and Evaluation.* Ablex Publishing Company, 1992.

[Dra86] S Draper. Display managers as the basis for user-machine communication. In D A Norman and S W Draper, editors, *[ND86]*, chapter 16, pages 339–352. Lawrence Erlbaum Associates, Publishers, Hilldale, New Jersey, 1986.

[DRW95] A. Dix, D. Ramduny, and J. Wilkinson. Interruptions deadlines and reminders: Investigations into the flow of cooperative work. Technical Report RR9509, School of Computing and Mathematics, University of Huddersfield, 1995.

[DW98] D F D'Souza and A Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach.* Addison Wesley, 1998.

[EH93] Frits L. Engel and Reinder Haakma. Expectations and Feedback in User-System Communication. *International Journal of Man-Machine Studies*, 39(3):427–452, 1993.

[EL96] Stephen G. Eick and Paul J. Lucas. Displaying trace files. *Software—Practice and Experience*, 26(4):399–409, April 1996.

[EN96] Margery Eldridge and William Newman. Agenda Benders: Modelling the Disruptions Caused by Technology Failures in the Workplace. In *Proceedings of ACM CHI 96 Conference on Human Factors in Computing Systems*, volume 2 of *SHORT PAPERS: Models of Work Practice (Short Papers Suite)*, pages 219–220, 1996.

[Eng97] R Englander. *Developing Java Beans.* O'Reilly, Cambridge, USA, June 1997.

[ES98] G Eisenhauer and K Schwan. An Object-Based Infrastructure for Program Monitoring and Steering. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, 1998.

[ESS92] S G Eick, J L Steffen, and E E Sumner. SeeSoft — A Tool for visualising software. *IEEE Transactions on Software Engineering*, 18:957–968, November 1992.

[FA93] Svend Frølund and Gul Agha. A Language Framework for Multi-Object Coordination. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 346–360, Kaiserslautern, Germany, July 1993. Springer Verlag, Lecture Notes in Computer Science. Vol. 707.

[Fau98] C Faulkner. *The Essence of Human Computer Interaction.* Prentice Hall, London, 1998.

[FFW88] Brad Hartfield Fernando Flores, Michael Graves and Terry Winograd. Computer systems and the design of organizational interaction. *ACM Transactions on Information Systems*, 6(2):153–172, April 1988.

[FHLS99] G Froehlich, H Jim Hoover, L Liu, and P Sorenson. Designing Object-Oriented Frameworks. In *[Zam99]*, chapter 25. 1999.

[FM95] Batya Friedman and Lynette Millett. "It's the Computer's Fault" – Reasoning about Computers as Moral Agents. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, volume 2 of *Short Papers: Agents and Anthropomorphism*, pages 226–227, 1995.

[FN97] BJ Fogg and Clifford Nass. How Users Reciprocate to Computers: An Experiment that Demonstrates Behavior Change. In *Proceedings of ACM CHI 97 Conference on Human Factors in Computing Systems*, volume 2 of *SHORT TALKS: A Melange*, pages 331–332, 1997.

[FP90] T Fawcett and F Provost. Activity Monitoring: Noticing Interesting Changes in Behaviour. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Diego, CA USA, August 15–18 1990. ACM.

[Fur86] George W. Furnas. Generalized Fisheye Views. In Marilyn M. Mantei and Peter Orbeton, editors, *Proceedings of the ACM Conference on Human Factors in Computer Systems*, SIGCHI Bulletin, pages 16–23. Association for Computer Machinery, New York, U.S.A., 1986.

[FvD82] J D Foley and A van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, Reading, Mass. London, 1982.

[Gar87] M M Gardiner. Principles from the psychology of memory. In *[GC87]*, chapter 5, pages 119–162. John Wiley & Sons, 1987. Part II. Episodic Memory.

[GC87] M M Gardiner and B Christie, editors. *Applying Cognitive Psychology to User Interface Design*, Chichester, 1987. John Wiley & Sons.

[GEC98] Nahum Gershon, Stephen G. Eick, and Stuart Card. Design: Information Visualization. *interactions*, 5(2):9–15, 1998.

[GGM97] Rachid Guerraoui, Benoît Garbinato, and Karim R. Mazouni. Garf: A Tool for Programming Reliable Distributed Applications. *IEEE Concurrency*, 5(4):32–39, October/December 1997.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1994.

[Gla98] Robert L. Glass. Loyal Opposition: Reuse: What's Wrong with This Picture? *IEEE Software*, 15(2):57–59, March / April 1998.

[Got98] E Gottesdiener. OO Methodologies. Process & Product Patterns. *Component Strategies*, 1(5):34–44, November 1998.

[GR93] J Gray and A Reuter. *Transaction Processing — Concepts and Techniques.* Morgan Kaufmann, San Francisco, California, 1993.

[Gru87] J Grudin. Social Evaluation of the user interface: Who does the work and who gets the benefit. In H-J Bullinger and B Shackel, editors, *INTERACT 1987. IFIP Conference on Human-Computer Interaction.*, Stuttgart, Germany, 1987. IFIP, Elsevier Science Publishers B.V.

[Gut99] Rhett Guthrie. The Business Case for Server Component Models. *Component Strategies*, 1(12):24–29, June 1999.

[Ham87] N Hammond. Principles from the psychology of skill acquisition. In *[GC87]*, chapter 6, pages 163–188. John Wiley & Sons, 1987.

[Han98] J Han. Characterisation of Components. In *[WCB88]*, 1998.

[Har98] P Harmon. Components and Objects. http://www.cutter.com/cds/cds9807.html, July 1998. Component Development Strategies. Monthly Newsletter from the Cutter Information Corporation.

[hCKBR97] Ed Huai hsin Chi, Joseph Konstan, Phillip Barry, and John Riedl. A Spreadsheet Approach to Information Visualization. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, Programming by Demonstration, pages 79–80, Banff Canada, October 14 - 17 1997.

[HG99] D Hinchcliffe and M J Gaffney. Components: Where are They? http://www.objectnews.com, January 1999. (19/1/99).

[Hit87] Graham J Hitch. Principles from psychology of memory. In *[GC87]*, chapter 5. John Wiley & Sons, 1987. Part I. Working Memory.

[HK93] Y. Huang and C. M. R. Kintala. Software implemented fault tolerance: Technologies and experience. In *Proceedings of 23rd Intl. Symposium on Fault-Tolerant Computing*, pages 2–9, Toulouse, France, June 1993. Also appeared as a chapter in the book Software Fault Tolerance, M. Lyu (Ed.), John Wiley & Sons, March 1995.

[HL95] Walter Hürsch and Cristina Videira Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, Massachusetts, February24 1995.

[Hub83] G P Huber. Cognitive style as a basis for MIS and DSS designs: much ado about nothing? *Management Science*, 29(5):565–597, 1983.

[ics98] *1998 International Workshop on Component-Based Software Engineering. ICSE98*, Kyoto, Japan, April 25-26 1998.

[JA84] G Johansson and G Aronsson. Stress reactions in computerized administrative work. *Journal of Occupational Behaviour*, 5:159–181, 1984.

[Jac73] J Jacobi. *The Psychology of C G Jung.* Yale University Press, New York, 1973.

[Jam96] Francis Jambon. *Erreurs et interruptions du point de vue de l'ingénierie de l'interaction homme-machine.* Phd thesis, Université Joseph Fourier, 1996.

[Jam98] F Jambon. Taxonomy for Human Error and Fault Recovery from the Engineering Perspective. In *International Conference on Human-Computer Interaction in Aeronautics (HCI-Aero'98)*, pages 55–60, Montreal, Canada, May 1998.

[Jam99a] J James. Design of the Kan Distributed System. Technical Report TRCS99-29, University of California, Santa Barbara, Santa Barbara, CA 93106, 1999.

[Jam99b] J James. *Reliable Distributed Objects: Reasoning, Analysis, and Implementation.* PhD thesis, UNIVERSITY OF CALIFORNIA, Santa Barbara, March 1999.

[Jam00] F Jambon. Personal communication, May 2000.

[Jer96] Dean F. Jerding. Visualizing Patterns in the Execution of Object-Oriented Programs. In *Proceedings of ACM CHI 96 Conference on Human Factors in Computing Systems*, volume 2 of *Doctoral Consortium*, pages 47–48, 1996.

[JLSU87] J Joyce, G Lomow, K Slind, and B Unger. Monitoring Distributed Systems. *ACM Transactions on Computer Systems*, 5(2):121–150, May 1987.

[JZTB98] C Jeffery, W Zhou, K Templer, and M Brazell. A Lightweight Architecture for Program Execution Monitoring. In *ACM SIGPLAN/SIGSOFT workshop on Progra Analysis for Software Tools and Engineering*, Montreal, Canada, June 16 1998. ACM.

[Kar98] D Kara. Build vs Buy: Maximizing the Potential of Components. *Component Strategies*, 1(1), July 1998. http://www.componentmag.com/.

[Kar99] D Kara. The Enterprise Java Beans Component Model. *Component Strategies*, 1(7):18–25, January 1999.

[Kea88] G Kearsley. *Online Help Systems. Design and Implementation.* Ablex Publishing Corporation, Norwood, New Jersey, 1988.

[KF90] David Kurlander and Steven Feiner. A Visual Language for Browsing, Undoing, and Redoing Graphical Interface Commands. In S.-K. Chang, editor, *Visual Languages and Visual Programming*, pages 257–275. Plenum, New York, 1990.

[KHB99] Murat Karaorman, Urs Hölzle, and John Bruno. jContractor: A Reflective Java Library to Support design by contract. Technical Report TRCS98-31, University of California, Santa Barbara. Computer Science., January 19, 1999.

[Kic96] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es):154–154, December 1996.

[Kie98] D Kiely. The Component Edge – An Industrywide Move To Component-Based development holds the promise of massive productivity gains. *Information Week*, (677), April 1998. www.techweb.com/se/directlink.cgi?IWK19980413S0001.

[KM99] Mik A. Kersten and Gail C. Murphy. Atlas: A Case Study in Building a Web-Based Learning Environment Using Aspect-oriented Programming. Technical Report TR-99-04, Department of Computer Science, University of British Columbia, March 31 1999. Wed, 07 Apr 1999 21:31:26 GMT.

[KMS+95] D. Kimelman, P. Mittal, E. Schonberg, P. F. Sweeney, Ko-Yang Wang, and D. Zernik. Visualizing the Execution of High Performance Fortran (HPF) Programs. In IEEE, editor, *IPPS '95: 9th International parallel processing symposium — April 25–28, 1995, Santa Barbara, CA*, International Parallel Processing Symposium, pages 750–759, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. IEEE Computer Society Press.

[KP95] M Kitajima and P G Polson. A comprehension-based model of correct performance and errors in skilled, display-based, human-computer interaction. *International Journal of Human-Computer Studies*, 43:65–99, 1995.

[KPS95] Harsha Kumar, Catherine Plaisant, and Ben Shneiderman. Browsing Hierarchical Data with Multi-Level Dynamic Queries and Pruning. Technical Report CS-TR-3474, HCIL Dept. of Computer Science, University of Maryland, March 1995.

[Kra88] Sacha Krakowiak. *Principles of Operating Systems.* MIT Press, Cambridge, 1988.

[Lew86] C Lewis. Understanding what's happening in system interactions. In D A Norman and S W Draper, editors, *[ND86]*, chapter 8, pages 171–186. Lawrence Erlbaum Associates, Publishers, Hilldale, New Jersey, 1986.

[Lew98] Scott M. Lewandowski. Frameworks for component-based client/server computing. *ACM Computing Surveys*, 30(1):3–27, March 1998.

[Lin91] K. Lindstrom. Breakdowns and other interruptions in VDT work as a source of stress in customer service and banking. In *Proceedings of the Fourth International Conference on Human-Computer Interaction*, volume 1 of *Congress I: Work with Terminals: HEALTH ASPECTS: WORKLOAD, STRESS AND STRAIN AND IRREGULAR WORKING HOURS; Causes and Measures of Stress*, pages 185–189, 1991.

[LL94] C. V. Lopes and K. J. Lieberherr. Abstracting Process-to-Function Relations in Concurrent Object-Oriented Applications. In *[TP94]*, pages 81–99, 1994.

[LLM91] X Lin, P Liebscher, and G Marchionini. Graphical Representations of Electronic Search Patterns. *Journal of the American Society for Information Science*, 42(7):469–478, 1991.

[LM88] P Leibscher and G Marchionini. Browse and analytical search strategies in a full-text CD-ROM encyclopedia. *School Library Media Quarterly*, Summer:223–233, 1988.

[LM94] Benoît Lemaire and Johanna Moore. An improved interface for tutorial dialogues: Browsing a visual dialogue history. In *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems*, volume 2 of *PAPER ABSTRACTS: Accessing and Exploring Information*, page 200, 1994.

[LN86] C Lewis and D A Norman. Designing for Error. In D A Norman and S W Draper, editors, *User Centred System Design. New Perspectives on Human-Computer Interaction*, chapter 20, pages 411–432. Lawrence Erlbaum Associates, Publishers, Hilldale, New Jersey, 1986.

[LS97] Gary T. Leavens and Murali Sitaraman, editors. *Proceedings of the First Workshop on the Foundations of Component-Based Systems, Zurich, Switzerland, September 26 1997*, September 1997.

[Mac91] Wendy E. Mackay. Triggers and Barriers to Customizing Software. In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, Group Use of Computing, pages 153–160, 1991.

[Mac99] Murdoch Mactaggert. Cbd, components and class libraries. *Application Development Advisor*, pages 14–17, Sept-Oct 1999.

[Man87] Ken Manktelow. Principles from the psychology of thinking and mental models. In *[GC87]*, chapter 4. John Wiley & Sons, 1987.

[Mar89] G Marchionini. Information-seeking strategies of novices using a full-text electronic encyclopedia. *Journal of the American Society for Information Science*, 50:54–66, 1989.

[MB99] T Merridenard and J Bird. Filling the gap. In *Management Today*. June 1999. Produced for Microsoft.

[McI68] M D McIlroy. Mass produced software components. In *[NR69]*, pages 88–98, 1968.

[MCLM90] Allan MacLean, Kathleen Carter, Lennart Lovstrand, and Thomas Moran. User-Tailorable Systems: Pressing the Issues with Buttons. In *Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems*, End User Modifiable Environment, pages 175–182, 1990.

[Mes98] J J Meserve. Application servers come into focus. Web Document, October 1998. Application Development Trends. www.adtmag.com.

[MG00] J McKirdy and P Gray. SUIT — Context Sensitive Evaluation of User Interface Development Tools. In *DSVIS'2000*, Limerick, Ireland, 2000.

[MGP60] G. A. Miller, E. Galanter, and K. H. Pribram. *Plans and the structure of behaviour*. Holt, Rinehart and Winston, New York, 1960.

[MGS92] A. Myka, U. Güntzer, and F. Sarre. Monitoring User Actions in the Hypertext System "HyperMan". In *ACM Tenth International Conference on Systems Documentation*, pages 103–113, 1992.

[Mic98a] Sun Microsystems. Enterprise Java Beans Specification. Web Document. URL: java.sun.com/products/ejb, March 1998.

[Mic98b] Sun Microsystems. javadoc - The Java API documentation Generator. Web Document, 1998. http://java.sun.com/products/jdk/1.2/docs/tooldocs/javadoc/.

[Mic98c] Sun Microsystems. JNDI: Java Naming and Directory Interface. http://java.sun.com/products/jndi/docs.html#11, January 1998. Web Document.

[Mic99] Sun Microsystems. java.lang.reflect. Web Document, 1999. http://java.sun.com/products//jdk/1.2/docs/api/java/lang/reflect/package-use.html.

[Mil94] Steve Miller. *Experimental Design and Statistics*. New Essential Psychology. Routledge, London and New York, second edition, 1994.

[MK93] M Masson and V De Keyser. Preventing human errors in skilled activities through a computerised support system. In *[SS93]*, 1993. volume II.

[MN86] Yoshiro Miyata and Donald A Norman. Psychological issues in support of multiple activities. In D A Norman and S W Draper, editors, *[ND86]*, chapter 13, pages 171–186. Lawrence Erlbaum Associates, Publishers, Hilldale, New Jersey, 1986.

[MNG87] C Marshall, C Nelson, and M M Gardiner. Design guidelines. In *[GC87]*, chapter 8, pages 221–278. John Wiley & Sons, 1987.

[Mos92] Vicky Mosley. How to Assess Tools Efficiently and Quantitatively. *IEEE Software*, 9(3):29–32, May 1992.

[MS99] H. Möessenböck and C. Steindl. The Oberon-2 reflection model and its applications. *Lecture Notes in Computer Science*, 1616:40–53, 1999.

[Mul93] S Mullender. *Distributed Systems.* Addison Wesley, Wokingham, second edition, 1993.

[ND86] D A Norman and S W Draper, editors. *User Centred System Design. New Perspectives on Human-Computer Interaction.* Lawrence Erlbaum Associates, Publishers, Hilldale, New Jersey, 1986.

[ND99] P A Nixon and S A Dobson. Objects, Components and the Virtual Enterprise. Technical Report TCD-CS-1999-07, Trinity College, Department of Computing Science, Trinity College, Dublin 2, Ireland, February 1999.

[Nic76] R Nickerson. On conversational interaction with computers. In *Proceedings of ACM/SIGGRAPH workshop*, pages 101–113, Pittsburgh, PA, 14–15 October 1976.

[Nie93] J Nielsen. *Usability Engineering.* AP Professional, Boston, 1993.

[Nor86] D Norman. Cognitive engineering. In D A Norman and S W Draper, editors, *[ND86]*, chapter 3, pages 31–62. Lawrence Erlbaum Associates, Publishers, Hilldale, New Jersey, 1986.

[Nor89] D Norman. The "problem" of automation: Inappropriate feedback and interaction, not "overautomation". Technical Report ICS Report 8904, Institute for Cognitive Science, University of California, San Diego, La Jolla, California, 92093, 1989.

[Nor94] D A Norman. *Things That Make Us Smart : Defending Human Attributes in the Age of the Machine.* Addison Wesley Publishing Company, 1994.

[Nor98] D A Norman. *The design of everyday things.* MIT Press, London, England, 98.

[NR69] P Naur and B Randell, editors. *Proceedings, NATO Conference on Software Engineering*, Garmish, Germany, October 1969. NATO Science Committee, Brussels (published as a book in 1976).

[NT91] M. A. Norman and P. J. Thomas. Informing HCI Design through Conversation Analysis. *International Journal of Man-Machine Studies*, 35(2):235–250, 1991.

[NT95] O Nierstrasz and D Tsichritzis, editors. *Object-Oriented Software Composition.* Prentice Hall, London, 1995.

[O'C99] A O'Callaghan. Full moon rising. *Application Development Advisor*, pages 59–61, May-June 1999.

[OF95] Brid O'Conaill and David Frohlich. Timespace in the workplace: Dealing with interruptions. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, volume 2 of *Short Papers: Workplaces and Classrooms*, pages 262–263, 1995.

[O'H94] Kenton O'Hara. Cost of Operations Affects Planfulness of Problem-Solving Behaviour. In *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems*, volume 2 of *INTERACTIVE POSTERS*, pages 105–106, 1994.

[OI94] H. Okamura and Y. Ishikawa. Object Location Control using Meta-level Programming. In *[TP94]*, pages 299–319, 1994.

[Ols87] J R Olsen. Cognitive Analysis of People's Use of Software. In *[Car87]*, chapter 10, pages 260–293. MIT Press, 1987.

[PAD+97] Tony Printezis, Malcolm P. Atkinson, Laurent Daynès, Susan Spence, and Pete Bailey. The Design of a new Persistent Object Store for PJama. In *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*, Half Moon Bay, CA, USA, August 1997.

[Par72] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Phi86] R J Phillips. Computer graphics as a memory aid and a thinking aid. *Journal of Computer Assisted Learning*, 2:37–44, 1986.

[PMR+96] Catherine Plaisant, Brett Milash, Anne Rose, Seth Widoff, and Ben Shneiderman. LifeLines: Visualizing Personal Histories. In *Proceedings of ACM CHI 96 Conference on Human Factors in Computing Systems*, volume 1 of *PAPERS: Interactive Information Retrieval*, pages 221–227, 1996.

[PQS96] Manuel A. Perez-Quinones and John L. Sibert. Negotiating User-Initiated Cancellation and Interruption Requests. In *Proceedings of ACM CHI 96 Conference on Human Factors in Computing Systems*, volume 2 of *SHORT PAPERS: Models*, pages 267–268, 1996.

[Pri99] J Pritchard. *COM and CORBA Side by Side. Architectures, Strategies and Implementations.* Addison Wesley, Reading, Massachusetts, 1999.

[PS92] Robert M. Poston and Michael P. Sexton. Evaluating and Selecting Testing Tools. *IEEE Software*, 9(3):33–42, May 1992.

[PSV94] Dewayne E. Perry, Nancy A. Staudenmeyer, and Lawrence G. Votta. People, Organizations, and Process Improvement: Two experiments to discover how developers spend their time. *IEEE Software*, 11(4):36–45, July 1994.

[PT88] M A Planas and W C Treurniet. The Effects of Feedback During Delays in Simulated Teletext Reception. *Behaviour and Information Technology*, 7(2):183–191, 1988.

[PvV99] M Porter and M van Vliet. Expand your server-side toolkit with EJB. Web Document. IT Architect. Sunworld, April 1999. www.sunworld.com/sunworldonline/swol-04-1999/swol-04-itarchitect.html.

[Raj99] G S Raj. A detailed comparison of enterprise javabeans (ejb) & the microsoft transaction server (mts) models. Web Document, May 1999. http://members.tripod.com/gsraj/misc/ejbmts/ejbmtscomp.html.

[Ras87a] J Rasmussen. Cognitive control and human error mechanisms. In *[RDL87]*. John Wiley and Sons, 1987.

[Ras87b] J Rasmussen. Reasons, causes and human error. In *[RDL87]*. John Wiley and Sons, 1987.

[RC94] Ramana Rao and Stuart K. Card. The Table Lens: Merging Graphical and Symbolic Representations in an Interactive Focus+Context Visualization for Tabular Information. In Beth Adelson, Susan Dumais, and Judith Olson, editors, *Proceedings of the Conference on Human Factors in Computing Systems*, pages 318–322, New York, NY, USA, April 1994. ACM Press.

[RDL87] J Rasmussen, K Duncan, and J Leplat, editors. *New Technology and Human Error.* New Technologies and Work. Ed: Bernhard Wilpert. John Wiley and Sons, Chichester, 1987.

[RE98] R Rock-Evans. *DCOM Explained.* Digital Press, Boston, 1998.

[RE00] Karen Renaud and Huw Evans. Javacloak: Engineering Java Proxy Objects using Reflection. In M Weber, editor, *NET.OBJECTDAYS 2000. Messekongresszentrum Erfurt, Germany*, October 9-12 2000.

[Rea87a] J Reason. A framework for classifying errors. In *[RDL87]*. John Wiley and Sons, 1987.

[Rea87b] J Reason. A preliminary classification of mistakes. In *[RDL87]*. John Wiley and Sons, 1987.

[Rea90] J Reason. *Human Error.* Cambridge University Press, 1990.

[Ren99] K V Renaud. Tracking activity at the user interface in a Java application. Technical Report TR-1999-33, Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow, G12 8RZ, April 1999.

[RFG+88] Brian J. Reiser, Patricia Friedmann, Jody Gevins, Daniel Y. Kimberg, Michael Ranney, and Antonio Romero. A Graphical Programming Language Interface for an Intelligent Lisp Tutor. In *Proceedings of ACM CHI'88 Conference on Human Factors in Computing Systems*, Visualization, pages 39–44, 1988.

[Ric91] G P Richardson. *Feedback Thought in Social Science and Systems Theory.* University of Pennsylvania Press, Philadelphia, 1991.

[RM93] George G. Robertson and Jock D. Mackinlay. The Document Lens. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, Visualizing Information, pages 101–108, 1993.

[Rog99] The Component Buyer's Guide. White Paper, March 1999. Rogue Wave Software Inc.

[RPMB96] A. Rizzo, O. Parlangeli, E. Marchigiani, and S. Bagnara. The management of human errors in user-centered design. *ACM SIGCHI Bulletin*, 28(3):114–119, 1996.

[RS97] Charles Rich and Candace L. Sidner. Segmented Interaction History in a Collaborative Interface Agent. In *Proceedings of the 1997 International Conference on Intelligent User Interfaces*, Planning Based Approaches, pages 23–30, 1997.

[RS99] E Roman and R Sessions. EJB vs COM+. Debate at Austin Foundation for Object Oriented Technology (AFOOT), July 13 1999. www.objectwatch.com/eddebate.htm.

[SA89] Lawrence M. Schleifer and Benjamin C. Amick, III. System response time and method of pay: Stress effects in computer-based tasks. *International Journal of Human-Computer Interaction*, 1(1):23–39, 1989.

[SBB96] Michael Spenke, Christian Beilken, and Thomas Berlage. FOCUS: The Interactive Table for Product Comparison and Selection. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, Papers: Information Visualization, pages 41–50, 1996.

[See98] K Seetharaman. The CORBA Connection. *Communications of the ACM*, 41(10):34–36, October 1998.

[Ses98a] R Sessions. *COM and DCOM. Microsofts Vision for Distributed Objects.* John Wiley and Sons, Inc, New York, 1998.

[Ses98b] R Sessions. The Convergence of TPMs and Components. Web Document, September 1998. www.objectwatch.com/converge.htm.

[Ses99] R Sessions. ObjectWatch Newsletter Number 22. Focus on Distributed Technology. Web Document, October 30 1999. www.objectwatch.com/issue22.htm.

[Ses00] R Sessions. *COM+ and the Battle for the Middle Tier.* Wiley, New York, 2000.

[SH92] M Siegle and R Hofmann. Monitoring Program Behaviour on SUPRENUM. In *International Conference on Computer Architecture. Proceedings of the 19th Annual International Symposium on Computer Architecture*, Queensland, Australia, May 19-21, 1992 1992. ACM.

[Shn86] B Shneiderman. *Designing the User Interface.* Addison-Wesley, Reading, Massachusetts, 1986.

[Shn98] Ben Shneiderman. *Designing the User Interface.* Addison-Wesley, Reading, Massachusetts, 1998.

[Sho98] K Short. Component-based development and object modeling. http://www.selectst.com, June 1998. (20 April 1999).

[Sid94] Candice L. Sidner. An Artificial Discourse Language for Collaborative Negotiation. In Barbara Hayes-Roth and Richard Korf, editors, *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 814–819, Menlo Park, California, 1994. American Association for Artificial Intelligence, AAAI Press.

[Sie98] J Siegel. OMG Overview: CORBA and the OMA in Enterprise Computing. *Communications of the ACM*, 41(10):34–36, October 1998.

[Sim69] Herbert A Simon. *The Sciences of the Artificial.* The M.I.T Press, Cambridge, Massachusetts, 1969.

[SKB99] P A Savage-Knepshield and N J Belkin. Interaction in Information Retrieval: Trends over Time. *Journal of the American Society for Information Science*, 50(12):1067–1082, 1999.

[SS93] G Salvendy and M J Smith, editors. *Advances in Human Factors/Ergonomics. Proceedings of the Fifth International Conference on Human-Computer Interaction, (HCI International '93)*, Orlando, Florida, August 8-13 1993. Elsevier, Amsterdam.

[SS98] Amanda Spink and Tefko Saracevic. Human-Computer Interaction in Information Retrieval: Nature and Manifestations of Feedback. *Interacting with Computers*, 10(3):249–267, 1998.

[SSTR93] Manojit Sarkar, Scott S. Snibbe, Oren J. Tversky, and Steven P. Reiss. Stretching the Rubber Sheet: A Metaphor for Viewing Large Layouts on Small Screens. Technical Report CS-93-39, Department of Computer Science, Brown University, Box 1910, Providence, RI 02912, September 1993.

[Sto94] G Storrs. A conceptualization of multiparty interaction. *Interacting with Computers*, 6(2):173–189, 1994.

[Str93] Robert Stroud. Transparency and reflection in distributed systems: position paper of the 5th ACM SIGOPS european workshop. *ACM Operating Systems Review, SIGOPS*, 27(2):99–103, April 1993.

[Str99] W Strigel. What's the problem: Labor Shortage or Industry Practices. *IEEE Software*, 16(3):52–54, May/June 1999.

[Suc87] L Suchman. *Plans and Situated Actions.* Cambridge University Press, Cambridge, 1987.

[SW89] J A Simpson and E S C Weiner, editors. *Oxford English Dictionary.* Clarendon Press, Oxford, second edition, 1989.

[SW98] D Sprott and L Wilkes. Component-based development. http://www.butlergroup.com/pubsframe, September 1998. (20 April 1999).

[Szy98] C Szyperski. *Component Software. Beyond Object Oriented Programming.* Addison Wesley, Harlow, England, 1998.

[TA92] G Torkzadeh and I E Angulo. The Concept and Correlates of Computer Anxiety. *Behaviour and Information Technology*, 11(2):99–108, 1992.

[Tay99] David A Taylor. The Keys to Object Technology. In *[Zam99]*, chapter 1. 1999.

[TB80] R N Taylor and I Benbasat. A critique of cognitive style theory and research. In *Proceeding of the First International Conference on Information Systems*, 1980.

[TB96] J G Trafton and D P Brock. Simplifying interactions with task model tracing. ACT-R Summer School, Psychology Department, Carnegie Mellon University, June 1996.

[Thi90] H Thimbleby. *User Interface Design.* Frontier. ACM Press, Addison Wesley Publishing Company, New York, 1990.

[Thi93a] M A Thies. Animated help as a sensible extension of a plan-based help system. In *[SS93]*, 1993. volume II.

[Thi93b] H Thimblebey. Combining systems and manuals. In J L Alty, D Diaper, and S Draper, editors, *People and Computers VIII HCI'93*, pages 479–88, 1993.

[Tho96] Richard C. Thomas. Long-Term Variation in User Actions. *ACM SIGCHI Bulletin*, 28(2):36–38, 1996.

[Tho97] A Thomas. Enterprise Java Beans. Server Component Model for Java. Patricia Seybold Group, Dec 1997.

[Tho98a] Anne Thomas. Enterprise JavaBeans technology. Server Component Model for the Java Platform. Web Document, 1998. http://java.sun.com/products/ejb/white_paper.html.

[Tho98b] Anne Thomas. Selecting Enterprise JavaBeans Technology. Prepared for WebLogic, Inc., July 1998. http://www.beasys.com/products/weblogic/server/papers.html.

[TJ93] Richard N. Taylor and Gregory F. Johnson. Separations of Concerns in the Chiron-1 User Interface Development and Management System. In Stacey Ashlund, Ken Mullet, Austin Henderson, Erik Hollnagel, and Ted White, editors, *Proceedings of the Conference on Human Factors in computing systems*, pages 367–374, New York, 24–29 April 1993. ACM Press.

[TK98] O Tallman and J B Kain. COM versus CORBA: A Decision Framework. Web Document, December 1998. www.quoininc.com/quoininc/COM_CORBA.html.

[TL91] Boon Wan Tan and Tak Wah Lo. The Impact of Interface Customization on the Effect of Cognitive Style on Information System Success. *Behaviour and Information Technology*, 10(4):297–310, 1991.

[TMdlADF99] Lourdes Tajes-Martines and Maria de los Angeles Diaz-Fondon. Systems Object Model (SOM). In *[Zam99]*, chapter 30. 1999.

[TN99] S Terzis and P Nixon. Component trading: The basis for a component-oriented development framework. In *4th International Workshop on Component-Oriented Programming (WCOP 99),(in conjunction with ECOOP'99)*, Lisbon, Portugal, 14 June 1999.

[TP94] Mario Tokoro and Remo Pareschi, editors. *Object-Oriented Programming, Proceedings of the 8th European Conference ECOOP'94. Lecture Notes in Computer Science*, volume 821, Bologna, Italy, July 1994. Springer Verlag.

[Tra91] D Travis. *Effective Color Displays. Theory and Practice.* Academic Press, London, 1991.

[Tuf90] Edward R. Tufte. *Envisioning Information.* Graphics Press, Cheshire, Connecticut, U.S.A., May 1990.

[Tul93] T S Tullis. Is user interface design just common sense? In *[SS93]*, 1993. volume II.

[TV99] S Taylor and J Vaughan. OTMs – ORBs for the enterprise. Web Document, February 1999. Application Development Trends. www.adtmag.com.

[Twe97] Lisa Tweedie. Characterizing Interactive Externalizations. In *Proceedings of ACM CHI 97 Conference on Human Factors in Computing Systems*, volume 1 of *PAPERS: Information Structures*, pages 375–382, 1997.

[VG94] Jean Vanderdonckt and Xavier Gillo. Visual Techniques for Traditional and Multimedia Layouts. In *Proceedings of the workshop on Advanced visual interfaces*, pages 95–104, Bari Italy, June 1994.

[VL99] J Vaughan and G Lawton. Application Servers in 1999: Persistent objects are knocking at the door. Web Document., May 1999. Application Development Trends. www.adtmag.com.

[vSBvL98] Rini van Solingen, Egon Berghout, and Frank van Latum. Interrupts: Just a Minute Never Is. *IEEE Software*, 15(5):97–103, September/October 1998.

[Wae89] Y Waern. *Cognitive Aspects of Computer Supported Tasks.* John Wiley & Sons, Chichester, 1989.

[War00] Colin Ware. *Information Visualization — Perception for Design.* Morgan Kaufmann, San Francisco, 2000.

[WC95] Wade Walker and Harvey G. Cragon. Interrupt Processing in Concurrent Processors. *Computer*, 28(6):36–46, June 1995.

[WCB88] ACM. *1998 International Workshop on Component-Based Software Engineering. Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Kyoto, Japan, April 25-26, 1998 1988. URL: http://www.sei.cmu.edu/cbs/icse98/papers/index.html.

[WD98] N Ward-Dutton. Componentware turns the corner. http://www.adtmag.com/pub/jul98/qa701.html, July 1998. Application Development Trends.

[WH88] D Wybranietz and D Haban. Monitoring and performance measuring distributed systems during operation. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 197–206, Santa Fe, USA, May 1988. ACM.

[Wil99] Shawn Willett. Cloudscape Woos VARs for Java Database. *Computer Reseller News 6-99*, June 1999. http://www.crn.com/search/display.asp?ArticleID=7017.

[WM99] Alan Wexelblat and Pattie Maes. Footprints: History-Rich Tools for Information Foraging. In *Proceedings of ACM CHI 99 Conference on Human Factors in Computing Systems*, volume 1 of *Foundations for Navigation*, pages 270–277, 1999.

[WS99] I. Welch and R. Stroud. From Dalang to Kava — the evolution of a reflective Java extension. *Lecture Notes in Computer Science*, 1616:2–21, 1999.

[WS00] Gillian M Wilson and Angela Sasse. Do Users Always Know What's Good for Them? Utilising Physiological Responses to Asses Media Quality. In Sharon McDonald,

Yvonne Waern, and Gilbert Cockton, editors, *Human Computer Interaction 2000. People and Computers XIV — Usability or Else! HCI2000*, Sunderland, United Kingdom, September 5-8 2000. Springer Verlag.

[WSA97] Ray Welland, Dag Sjøberg, and Malcolm Atkinson. Empirical Analysis based on Automatic Tool Logging. In *Empirical Assessment in Software Engineering. EASE'97*, University of Keele., 24-26 March 1997.

[YC93] C Yang and P Carayon. Effects of computer system performance and job support on stress among office workers. In *[SS93]*, 1993. volume I.

[YO94] Toshiya Yoshimune and Katsuhiko Ogawa. Graphical Feedback System to Effectively Support User's Task. In *Proceedings of the Human Factors and Ergonomics Society 38th Annual Meeting*, volume 1 of *COMPUTER SYSTEMS: Usability [Lecture]*, pages 345–349, 1994.

[Zak92] D Zakay. The influence of computerized feedback on overconfidence in knowledge. *Behaviour & Information Technology*, 11(6):329–33, 1992.

[Zam99] S Zamir, editor. *Handbook of Object Technology*. CRC Press, Boca Raton, 1999.

[ZBF+92] D Zapf, F C Brodbeck, M Frese, H Peters, and J Prümper. Errors in working with office computers: A first validation of a taxonomy for observed errors in a field setting. *International Journal of Human-Computer Interaction*, 4(4):311–339, 1992.