



A human-machine interaction tool set for Smalltalk 80.

SPALL, Roger P.

Available from the Sheffield Hallam University Research Archive (SHURA) at:

<http://shura.shu.ac.uk/20389/>

A Sheffield Hallam University thesis

This thesis is protected by copyright which belongs to the author.

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Please visit <http://shura.shu.ac.uk/20389/> and <http://shura.shu.ac.uk/information.html> for further details about copyright and re-use permissions.

SHEFFIELD CITY
POLYTECHNIC LIBRARY
POND STREET
SHEFFIELD S1 4WD

TELEPEN

100264958 7



13307
Sheffield City Polytechnic Library

REFERENCE ONLY

ProQuest Number: 10701035

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10701035

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

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

A HUMAN-MACHINE INTERACTION TOOL SET FOR SMALLTALK 80.

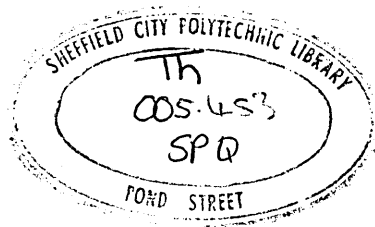
by

Roger Paul Spall, BSc.

**A thesis submitted in partial fulfilment of the
requirements of the Council for National Academic Awards
for the degree of Doctor of Philosophy**

**Sponsored by Sheffield City Polytechnic, in collaboration with MRC/SERC
Social and Applied Psychology Unit, University of Sheffield.**

May 1990



Contents.

Contents.....	i
List Of Figures.....	vi
Acknowledgements.....	vii
Abstract.....	viii

Chapter One.

Introduction and Thesis Outline.....	1
1.1. Introduction.....	1
1.2. Thesis Outline.....	3

Chapter Two.

Software Related Influences Affecting User Acceptance of Computer

Software Applications.....	6
2.1. Introduction.....	6
2.2. Simplicity.....	6
2.3. Consistency.....	7
2.4. Integration.....	9
2.4.1. Modes.....	11
2.5. Metaphor.....	13
2.6. Interaction Styles.....	15
2.6.1. Command and Natural Language.....	15
2.6.2. Menus.....	17
2.6.3. Direct Manipulation.....	18
2.6.4. Combinations.....	19
2.6.5. Dialogue Control and Specification.....	20
2.6.6. Style Guides.....	20
2.7. Error Handling.....	21
2.8. Documentation and Tutorials.....	25
2.9. Interface Separation.....	28
2.10. Interface Ergonomics.....	29
2.11. Summary.....	32

Chapter Three.

The Application of Artificial Intelligence to User Interface Design.....	34
3.1. Introduction.....	34
3.2. Additional Intelligent Interface Modules.....	34
3.2.1. Intelligent Help Systems.....	34
3.2.2. Models.....	38
3.2.2.1. User Model.....	39
3.2.2.2. Application Model.....	45
3.2.2.3. Real World Model.....	46
3.2.2.4. Summary.....	46
3.2.3. Adaptive Interfaces.....	48
3.2.4. Planning Aids.....	51
3.2.5. General Architecture for an Intelligent Interface.....	53
3.3. Interface Classification.....	56
3.4. Approaches to Interface Design.....	58
3.4.1. Requirements For Good Interface Design.....	60
3.4.2. Graphics Environment Manager.....	63

3.4.3. The Model View Controller Mechanism used in Smalltalk 80.....	65
3.5. Summary.....	67
 Chapter Four.	
Experience With Other Influences which Affect User Acceptance of Computer Systems.....	69
4.1. Introduction.....	69
4.2. The Working Library System.....	70
4.2.1. Overview.....	70
4.2.2. Library System Description.....	71
4.2.2.1. Database structure.....	71
4.2.2.2. User Interface.....	72
4.2.3. Library Investigation Results.....	73
4.2.3.1. Initial Notebook Investigation.....	73
4.2.3.2. Initial Interviews.....	74
4.2.3.3. Further Interviews.....	75
4.2.4. Summary.....	79
4.3. The Influence of Systems Analysis and Design upon User Acceptance of Computer Systems.....	81
4.3.1. Summary.....	87
 Chapter Five.	
An Investigation into the Quantitative User Modelling of User Interactions for the purpose of Predicting User Expertise.....	89
5.1. Introduction.....	89
5.2. Overview of the Proposed Quantitative User Model.....	92
5.3. Structure of the Proposed Quantitative User Model.....	93
5.4. Functioning of the Proposed Quantitative User Model.....	96
5.5. Evaluation and Conclusions.....	100
5.6. Summary.....	102
 Chapter Six.	
An Object Oriented User Interface Management System, and Integrated Interface Design Tool-set.....	103
6.1. Introduction.....	103
6.2. Overview of Object Oriented Programming.....	103
6.3. The Smalltalk 80 Programming Language and Environment.....	106
6.4. The User Interface Management System and Software Architecture.....	107
6.4.1. The Pluggable View Controller Mechanism.....	108
6.4.2. Communication Between Objects and Pluggable View Controllers.....	110
6.4.3. Defining a Direct Manipulation Interface for an Object.....	115
6.4.4. Part Pluggable View Controllers.....	116
6.4.5. Interaction Pluggable View Controllers.....	117
6.4.5.1. Internal Interaction Pluggable View Controller Structure.....	119
6.4.5.2. External Interaction Pluggable View Controller Description.....	123
6.4.6. Communication Between Interaction Pluggable View Controllers.....	127

6.4.7. Interaction Pluggable View Controller Cursors.....	129
6.4.8. Interaction Pluggable View Controller Multiple Linkage Slots.....	130
6.4.9. Default Part Pluggable View Controllers.....	132
6.4.10. Part Pluggable View Controllers and Part Hierarchies.....	133
6.4.11. Special Part Pluggable View Controllers.....	137
6.4.12. Interactive Creation of Smalltalk Objects.....	138
6.4.13. Construction and Interaction Menus.....	139
6.4.14. Extended Lean Cuisine Hierarchic Menus.....	139
6.5. The Part Pluggable View Controller Tool-set.	141
6.6. Summary.	142

Chapter Seven.

The Support of Part Hierarchy Mechanisms in an Object Oriented

Language.	144
7.1. Introduction.	144
7.2. What is a Part Hierarchy, and Why is it Needed ?	144
7.3. Difficulties with Object Oriented Languages and Part Hierarchies.	146
7.4. Solving the Problem.....	147
7.5. A Solution Implemented in Smalltalk 80.....	150
7.6. An Improved Solution.....	152
7.7. Summary.	152

Chapter Eight.

Critical Evaluation and Assessment of the Proposed User Interface

Management System and Smalltalk 80.....	154
8.1. Introduction.	154
8.2. Evaluation of the Underlying User Interface Management System Software Architecture.	154
8.2.1. Advantages of Interface Separation.....	155
8.2.2. Software Requirements for Interface Separation.....	156
8.2.2.1. The Number of Components required.	157
8.2.2.2. Application Function Set and its Preconditions.	159
8.2.2.3. Application and Interface States.	161
8.2.2.4. Component Communication Requirements.	164
8.2.2.5. Separation Controller.....	166
8.2.2.6. Interface Defaults.....	167
8.2.3. Interface Separation Design Constraints.....	167
8.2.3.1. Application Independence.....	167
8.2.3.2. General Application Design Constraints.	168
8.2.3.3. General Interface Design Constraints.	168
8.2.3.4. Constraints imposed by the Separate Application on the User Interface.	169
8.2.3.5. User Interface Constraints imposed on the Application.....	169
8.2.3.6. Summary.	169
8.2.4. Conclusions.....	170
8.3. Evaluation of the Object Oriented Paradigm and Smalltalk 80 Programming Language.	171

Chapter Nine.	
Suggested Further Work.	176
9.1. Introduction.	176
9.2. Extensions to existing User Interface Management System Implementation.	176
9.2.1. The User Interface Management System Architecture and Tool-Set.	176
9.2.2. Quantitative User Model.	179
9.2.3. Part Hierarchies.	180
9.3. Further User Interface Management System Implementation.	180
9.4. Smalltalk 80 Programming Language Extensions.	181
9.5. Further Systems Analysis and Design Work.	182
 Chapter Ten.	
Final Conclusions.	184
 Bibliography	187
 Appendix A.	
Library Notebook Statements.	230
A.1. Subject Two.	230
A.2. Subject Four.	231
A.3. Subject Five.	232
 Appendix B.	
An Overview of the Smalltalk 80 Programming Language and Environment.	234
 Appendix C.	
Object Oriented Quantitative User Model Source Code.	239
 Appendix D.	
Smalltalk 80 Example Object Code.	250
 Appendix E.	
Extended Backus Naur Formats.	267
E.1. General Syntax.	267
E.2. External PVC Slot Description.	267
E.2. Part PVC Description.	268
E.2. Extended Lean Cuisine Description.	269
 Appendix F.	
Interaction Pluggable View Controller Library.	271
 Appendix G.	
Class Hierarchy for Object Oriented User Interface Management System Implementation.	287
 Appendix H.	
Example Interfaces and Associated Code Generated by the User Interface Management System.	291

Appendix I.

User Interface Management System Tool-Set Documentation.....	345
I.1. Creating New Part Pluggable View Controllers.....	345
I.2. Default Construction Menus.....	346
I.3. Adding Interaction Pluggable View Controllers.....	348
I.4. Adding Further Part Pluggable View Controllers.....	350
I.5. Aligning Pluggable View Controllers.....	350
I.6. Linking Interaction Pluggable View Controllers to one another.....	351
I.7. Modifying Pluggable View Controller Size.....	351
I.8. Closing Pluggable View Controllers.....	351
I.9. Modifying and Reviewing Pluggable View Controller Linkage Slots using the Inspector Window.....	352
I.10. Code Generation.....	352
I.11. Setting Default Part Pluggable View Controller.....	352
I.12. Changing Part Pluggable View Controllers.....	353
I.13. Spawning Part Pluggable View Controllers.....	353
I.14. Building Extended Lean Cuisine Hierarchic Menus.....	353

List Of Figures

Figure 2.6.2 - Hierarchic Menus.....	17
Figure 2.7 - Error Classification Methodology.....	23
Figure 3.2.1. Traditional Application Configuration	35
Figure 3.2.2.1a - User Classification.....	40
Figure 3.2.2.1b - Two General User Models.....	42
Figure 3.2.2.1c - Example Transition Diagram for MacWrite.....	43
Figure 3.2.3 - Adaptive Intelligent Dialogues Architecture.....	49
Figure 3.2.5 - Complete UIMS Architecture.....	54
Figure 3.3 - Classification of Interfaces.....	58
Figure 3.4.2 - GEM Architecture.....	64
Figure 3.4.3 MVC Concept.....	67
Figure 4.3 - Design Role Analogy.....	87
Figure 5.3 - Quantitative User Model Structure.....	93
Figure 5.4a - Usage Relationship Between Different Conceptual Levels.....	98
Figure 5.4b - User Learning Curve.....	99
Figure 6.2 - Object Class Definition and Instances.....	105
Figure 6.4.1 - Pluggable View Controller Mechanism.....	109
Figure 6.4.2a - Pluggable View Controller Communication.....	111
Figure 6.4.2b - Application / User Interface Communication.....	112
Figure 6.4.4 - Part Pluggable View Controller Architecture.....	116
Figure 6.4.5 - Interaction Pluggable View Controllers Model, and connection to a Part Pluggable View Controllers.....	118
Figure 6.4.5.1 - Interaction Pluggable View Controller, Part Pluggable View Controller, and screen Co-ordinate Systems.....	122
Figure 6.4.5.2 - Example Interaction Pluggable View Controllers.....	125
Figure 6.4.10 - Combining Part Pluggable View Controllers.....	136
Figure 7.2 - An example Part Hierarchy.....	145
Figure 8.2.2.1 - Two Approaches to Interface Separation.....	157
Figure 8.2.2.3 - Comparison of Switch Interaction Pluggable View Controllers.....	162
Figure 8.2.3.6 - Component Stability Within Separable Architecture.....	170
Figure 9.2.1 - Further Separation Within a PVC.....	177
Figure B1 - Example Smalltalk Class.....	235

Acknowledgements.

This project was funded under grant number 8631820X by the Science and Engineering Research Council.

I would like to thank my supervisor Bob Steele, and Director of studies Professor F. Poole for their help, and advice during the period leading up to this report.

I would like to thank my wife Elaine, for all of her moral support and encouragement during the project.

I would like to thank the MRC/SERC Social and Applied Psychology Unit at Sheffield University, the Human Factors Laboratory at ICL, Bracknell, and British Telecom Research Laboratories for their assistance with specific components of the research.

I would also like to thank Ian Morrey, and Dr Jawed Siddiqi who have also given their help and advice regarding the contents of this thesis.

Abstract.

This research represents an investigation into user acceptance of computer systems. It starts with the premise that existing systems do not fully meet user requirements, and are therefore rejected as 'difficult to use'. Various problems and influences affecting user acceptance are identified, and improvements are suggested. Although a broad range of factors affecting user acceptance are discussed, emphasis is given to the impact of actual computer software.

Initially, both general and specific user interface software influences are examined, and it is shown how these needs can be met using new software technology. A new Intelligent Interface architecture model is presented, and comparisons are made to existing interface design approaches.

Secondly, the role of empirical work within the field of Human Computer Interaction is highlighted. An investigation into the usability and user acceptance of a large working library database system is described, and the results discussed. The role of Systems Analysis and Design and its effect upon user acceptance is also explored. It is argued that despite improvements in interface technology and related software engineering techniques, a software application is also a product of the Systems Analysis and Design process. Traditional Systems Design approaches are examined, and suitable improvements suggested based upon experience with emerging separable software architectures.

Thirdly, the research proceeds to examine the potential of Quantitative User Modelling, and describes the implementation of an example object oriented Quantitative User Model. This is then evaluated in order to determine new knowledge, concerning the major issues surrounding the potential application of user modelling to interface design.

Finally, attention is given to the concept of interface and application separation. An object oriented User Interface Management System is presented, and its implementation in the Smalltalk 80 programming language discussed. The proposed User Interface Management System utilises a new software architecture which provides explicit user interface separation, using the concept of a Pluggable View Controller. It also incorporates an integrated design Tool-set for Direct Manipulation interfaces. The proposed User Interface Management System and software architecture represents the major contribution of this project to the growing body of Human Computer Interaction research. In particular, the importance of explicit interface separation is established, and the proposed software architecture is critically evaluated to determine new knowledge concerning the requirements, constraints, and potential of proper user interface separation. The implementation of an object oriented Part Hierarchy mechanism is also presented. This mechanism is related to the proposed User Interface Management System, and is critically evaluated in order to add to the body of knowledge concerning object oriented systems.

Introduction and Thesis Outline.

1.1. Introduction.

Due to its reduced cost and improved power, computer technology is becoming more widely used by people from a large range of backgrounds, and with different experience. As this varied user population expands, so do their requirements and expectations of new computer systems. Historically, computer systems only needed to work properly and efficiently in order to be acceptable. It was not unusual to employ and train specialist computer staff to use complicated computer systems. However, today's computer systems require a greater emphasis upon ease of use of the final system by both experienced, and inexperienced users. Many factors influence a systems 'usability', and research into these factors falls under the title of Human Computer Interaction.

Human Computer Interaction deals with the problems and issues affecting the actual interaction of a user with a computer system [Rasmussen, J:1987]. Its primary goal is to improve the acceptance of new computer systems by the user. Research in this area encompasses a vast number of topics, ranging from the effect of social and organisational factors [Dray, S.M:1987], [Grudin, J:1987] through to the effects of different software [Gould, J.D:1987] and hardware components [Hulme, C:1986].

A software system is a product of both Systems Analysis and Systems Design [Yau, S.S:1987], and its usage should always be viewed in the context of the personal motivations and work environment of the user [Whiteside, J:1986]. Problems with software systems which are rejected as unacceptable by the user, can often be traced to insufficient Systems Analysis, poor design and implementation, or the work environment of the user. These influences are closely related, and are often confused. Insufficient Systems Analysis will result in a system design which does not match the expectations or requirements of the user. Similarly, poor design and implementation will result in a system which does not match the system specification. Systems Analysis and design are related in many ways, and are dependant upon each other. The availability of specific design and implementation tools and techniques affects the type of analysis tasks performed, the decisions which are made during the analysis process, and the information which is

gathered. The accuracy and detail of the system specification generated by the Systems Analysis process also affects the precision of the final system design and implementation. Other influences related to the work environment and personal motivations of the user affect the final acceptance of software systems. These influences include physical stress, workload, organisational and managerial factors, and the personal prejudices and partiality of a user.

Certain factors affecting the acceptance of software systems are easier to control and improve than others. Due to their theoretical basis, enhancements to Systems Analysis and Design methods are possible, and therefore attract a growing research interest. Several major areas for improvement can be readily identified [Balzert, H:1987]. These include better software design features, advanced software architectures and design tools, and modified Systems Analysis and Design methods, which place emphasis upon designing systems for the user.

Some features of software systems generally affect all users in the same way, for example, consistency and integration. However, due to the variety of computer users, in the majority of cases it is only possible to identify software features which are 'user friendly' for specific types of user. Similarly, features can be identified which may be 'user unfriendly' for other user types. Unfortunately, the terms 'user friendly' and 'user unfriendly' are incorrectly used as a means of classifying individual computer systems. These terms are meaningless unless used in the context of individual software features, individual user requirements, the Systems Analysis and Design method employed, and the environment in which the software system is being used.

To facilitate the design of 'usable' software by software engineers, it is proposed that software systems should be divided into two separate components, the application and the user interface. The application represents the functionality of the software system. The user interface is defined as the software through which information flows between the user and the application. Information flows in two directions, with users specifying their information requirements and also viewing the application results. These two components affect the 'usability' of software in different ways. The use of separation should therefore enable effort to be concentrated in addressing these different affects. Other advantages also arise from the use of separation, including the facilitation of multiple user interfaces for a single application. Computer hardware also affects the working of a user

interface - for example, different keyboard layouts, the use of mouse or tracker ball pointing devices, high resolution colour graphics terminals, voice recognition and generation hardware [Burrough:1983], [Hagelberger, D.W:1983]. However, while acknowledging the effects of hardware, this research concentrates specifically upon the influences of software upon user acceptance of computer systems.

The discipline of Software Engineering traditionally emphasises the importance of designing software systems which are reliable, portable, easy to maintain, reasonably priced, delivered on time and perform well [Bell, D:1987]. The software architecture model underlying traditional implementation languages and systems, is also specifically aimed at meeting these criteria. The significance of designing software which is 'easy to use' is often ignored, and therefore remains unsupported within the underlying software architecture. The need for new software architectures is necessary if improvements to the 'usability' of future software systems are to be made. In particular, attention must be given to the role of Artificial Intelligence as a mechanism for automatically matching user interface software features to individual users. New User Interface Management Systems must also support the concept of explicit interface separation, and provide integrated tools to enable the designer to implement 'usable' user interfaces.

Empirical work plays a major role in the field of Human Computer Interaction research [Galer, M:1987]. Interaction with a computer system is an observable phenomenon, which can only be properly analysed using practical experimentation. Theoretical Human Computer Interaction research must therefore be based upon, or proven, using controlled experiments enabling the evaluation of users actually using real computer systems. This type of evaluation may be included as part of the regular Systems Analysis and Design processes, or may be restricted to the 'Human Factors' research laboratory.

1.2. Thesis Outline.

The research described begins with the premise that many existing computer systems do not meet user requirements or expectations, and are therefore unacceptable to the user. Chapter two identifies the major causes for the rejection of computer systems, and suggests possible solutions. Emphasis is given to the effect of software elements, and an extensive list of influential

software factors is presented. Where appropriate, improvements to existing software technology, and in particular the user interface, are also described.

The research maintains that the potential of software improvements is often dependent upon individual user preferences, and Artificial Intelligence is presented as a mechanism for matching interface features to individual users. Chapter three describes the application of Artificial Intelligence to the user interface. Specific interface design requirements are identified, an outline design for a new Intelligent Interface is given, and its individual modules described. This chapter also highlights the need for improved approaches to interface design, and discusses alternative methods, and their associated tools.

Chapter four relates the background work discussed in chapters two and three to the results of a detailed investigation into a large library database computer system, and its users. This chapter identifies the wider influences affecting user acceptance of computer systems, and stresses the importance of good Systems Analysis and Design methods.

Chapters five, six, and seven describe the Smalltalk 80 implementation work which was completed as part of this project. These chapters represent the original element of this research. Chapter five examines the application of Quantitative User Modelling, and details an example object oriented Quantitative User Model, which is then evaluated. Chapter six presents the design of a new User Interface Management System and its underlying software architecture. An integrated interface design Tool-set is also described. Chapter seven presents the design of an object oriented Part Hierarchy mechanism which is related to the work presented in chapter six. This is again evaluated, and alternative approaches discussed.

The issue of explicit interface separation is discussed in chapter eight, which critically evaluates the User Interface Management System, and software architecture proposed in chapter six. This chapter examines in detail how separation can be achieved, its potential, and its constraints.

Chapter nine describes further investigations and enhancements which may be the subject of future research, and finally, chapter ten presents a summary of the project's conclusions. This identifies the major research contributions to the existing body of knowledge concerning Human Computer Interaction, and object oriented systems.

Appendix A contains a list of statements collected during the Library System investigation. Appendix B provides a detailed overview of the salient Smalltalk 80 language features which are relevant to the proposed User Interface Management System implementation. Appendix C contains the Smalltalk 80 source code for the proposed Quantitative User Model implementation. Appendix D contains the Smalltalk 80 code for the object oriented examples used throughout the thesis. Appendix E presents the relevant Extended Backus Naur Form syntax for the work described in Chapter six. Appendix F describes the library of Interaction Pluggable View Controllers implemented to demonstrate the potential of the proposed User Interface Management System. Appendix G contains the Class hierarchy for the proposed User Interface Management System implementation. Appendix H contains example direct manipulation interfaces generated by the interface design Tool-set, and the Smalltalk 80 code generated by the proposed User Interface Management System. Finally, appendix I contains the documentation for the Tool-set presented in chapter six.

Chapter Two.

Software Related Influences Affecting User Acceptance of Computer Software Applications.

2.1. Introduction.

This chapter examines how various software related influences can affect user acceptance of computer systems. It identifies the major influences, and suggests methods for improving the quality of existing software. A distinction is made between the functionality of the application, and the user interface through which it is used. Inevitably, user acceptance of a complete software application depends equally upon both a functionally correct application, and a well designed interface. Certain software applications make this separation more explicit than others, and this chapter develops the argument in favour of explicit user interface separation.

2.2. Simplicity.

Current software is often complex and technically biased, tending towards as many features as possible, allowing the user to do more things with greater power. However, this trend is usually followed at the cost of simplicity, with most software being difficult to use and understand [Dean, M:1983].

The increasing power of computer technology will undoubtedly be utilised to provide more complex applications. However, the way in which this complexity is presented to the user needs careful consideration [Kornwachs, K:1987]. Novice users using an application for the first time are typically concerned with learning basic functions and tasks. Only as they become more familiar with the application, will they move onto more complex functions. Simplicity in this respect does not refer to the extent of application functionality, but rather to the way in which this functionality is presented to the user [Clark, I.A:1987]. Whenever possible, functional complexity should be hidden.

Progression by the user with a particular system is often hampered by errors made with commands or concepts they discover accidentally. They may become confused by the error and accordingly assume that the rest of the system is also difficult to use. Such errors and misconceptions at an early stage quickly reduce the confidence of a user in that system [Carroll,

J.M:1988]. Also, users are typically discouraged by both the sheer functional 'size' of an application, and having to 'navigate' this functional set in order to perform what are often trivial tasks. This seems to suggest the need for organising, and classifying application functions according to their complexity, which would enable function sub-sets to be automatically created by the interface according to the level of expertise needed to use them. The complete functionality of an application could therefore be hidden from a novice user [Spall,R.P:1987]. Function sub-sets may also prevent novice users 'getting lost' within a software system, and from accidentally using irreversible tasks, such as deletion. Whether expert functions are completely hidden, or shown to the user (who is prevented from using them) is an issue which lies outside of the scope of this research. An application should at least enable a novice user to use the basic function set immediately, with relative ease [Karat, J:1987]. It may in fact be the case that the extra facilities associated with expert functions encourage a novice user to become more expert. Simplicity also applies to other software influences including for example, length and content of error messages, advice, prompts, and screen layout.

2.3. Consistency.

A consistent application uses operation and object formats and structures that do not contradict one other, but support and cross reference each other. Consistency affects the relationship between the expectations of a user of the application, and the actual application itself [Kellog, W.A:1987]. For example, system errors always appearing in red at the bottom of the screen, which leads users to expect all error messages to appear there, and anything that appears there to be an error message. Any inconsistencies will cause problems, with the user having to make adjustments to the way they view the system. Another example is a <delete> command, which may apply in many different parts of the system. In each case it should have an identical format; e.g. DELETE <object name>, it should not be called ERASE in one part, or have a different format elsewhere.

Some further examples of consistency include :-

- naming conventions, e.g. all word processor text file names have a file extension of type .WS
- screen layouts, where titles always appear in a certain place and format

- icons should have identical meaning and representation between systems, or parts of systems.

Typically, one particular user may prefer, or use, different naming conventions, and styles to another. Consistency does not address this problem of individuality. However, individual user preferences must be used consistently throughout an entire software system.

Inconsistency may occur within both the user interface and application functionality. However, it may be possible for a separated user interface to present inconsistent application functions in a consistent form to the user. For example, a separated interface may provide a set of consistent user interactions which, unbeknown to the user, are mapped onto inconsistent application functions.

Consistency can be improved by the use of formal Systems Analysis and Design methods, and by the use of consistent design and implementation tools [Sharrat, B.D:1987]. Such methods and tools are particularly important where large numbers of people are working on the same application. Effectively, they force individual designers to follow certain agreed standards. Consistency applies equally to other areas of concern such as the user interface, documentation, and error messages.

The use of explicit software guide-lines provides a useful aid to consistency [NTIS:1987]. Such guide-lines specify standards which apply to the entire application, and usually take the form of internal printed documents. However, if these guide-lines are described in a computer readable form, a User Interface Management System could automatically interpret them and generate the appropriate user interface. Individual user preferences could then be handled by simple modification and re-compiling.

Consistency is difficult to achieve, and the consistency seen on the surface is often affected by deep structural consistency within software. Generally speaking there are two major goals for system developers :-

- consistency within software
- consistency between different software applications.

Consistency should enhance the confidence of a user in a system, as it responds in a predictable way. The opposite is also true, as an inconsistent

system may cause mistrust. Consistency is a vital component of any interface which is to be accepted and easily used. A minimum requirement therefore, is consistency within a software package. Consistency across different software packages is more difficult to attain, and a trade-off must often be made between consistency and 'software enrichment', using specialised application features or functions.

Consistency between software houses is an issue affected by many influences including :-

- copyright laws/controls and intellectual property rights
- fine tuning of software to hardware
- competition.

However, if consistency between products were attained, users could move from one application to another, taking with them more of their previous knowledge and experience, e.g. similar command names, icon shapes, and menu/screen layouts. If one company manufactures many software packages then there should be no problem in maintaining consistency across all of these packages. As regards consistency between software houses, the problem is more deeply rooted in business objectives. It is often in the interest of a company for its products to be inconsistent with another companies. This is more difficult to solve, and is beyond the scope of this research.

2.4. Integration.

Integration is the combination of simple tasks in order to complete a larger more complex task, and implies a high degree of functionality within these simple tasks. In our everyday life integration is taken for granted [Chang, D:1983]. However, many computer users pose intricate and complex tasks that defy easy computerisation. Given the benefits of complete software integration, these tasks become simple to solve by combining the use of several software applications. For example, the integration of a Word Processor, and Graphics system to provide a Desk Top Publishing system.

Integrated software must satisfy both human and machine requirements [Brown, M.J. 1983]. The human requirement is that information will pass effortlessly from one application program to another within the context of

user tasks. The machine requirement, or 'performance view' is that user information will be stored, shared and retrieved efficiently.

The crux of the problem is that individual applications often require unique data structures to work at maximum efficiency. But unfortunately, the more unique the data structures, the harder it is to exchange data. This is analogous to the problems of people with different abilities who must work together. For example, a group of programmers, engineers, accountants and product managers, who specialise in each of their respective areas and collaborate to achieve company goals. The problem occurs when a project requires the interaction of two or more of these groups. Although each person is competent in a given field, some proficiency in the other fields is needed for a successful project. If we substitute the unique abilities of a worker for the unique data structures of our program, we can see that the problem of information flow exists in both areas.

Components within a system should be fully integrated, e.g. spell checking, reformatting, searching, and replacing within a Word Processor [Paul, D.W:1987]. Systems should also be integrated with other systems. For example, Word Processors should be able to take input from other systems such as Accounting packages and Spread-sheets. They should also be able to generate useful output for other systems such as Mailing systems. This integration poses two major problems. These are, firstly, the technical problem of swapping information between two different software packages, and secondly, the conceptual problem of representing this integration to the user [Dirlich, G:1987].

Integration is assisted by a distinct separation between information, and the tasks which can use the information. The information can be moved between applications, and viewed or used differently in each. For example, a picture may appear as an icon to a file handling system, a certain size box to a word processor, or an actual detailed drawing to a graphics editor. Similarly, a user may create an initial picture using a painting package, move to a specialised drawing package to add certain features, and finally move to a word processor, where the picture is placed between certain text items.

Ideally software integration should model the way in which humans integrate knowledge and applications so as to aid ease of use [Vandor, S:1983]. The three most useful forms of integration are outlined below.

These are placed in order of 'naturalness', that is, how well they match human methods of integration.

Note Pad.

This method incorporates a Global Notebook to which information can be written to or read from at any time. The source of this information is the current screen or task. The destination of any information read from the notebook is also the current screen or task. The contents of this notebook are maintained between sessions, and can be edited, using cut, copy and paste functions.

Buffering Techniques.

This method allows users to mark a block of text on the current screen, and then make a copy of this block. The copy can then be inserted anywhere on future screens. Only one block can be copied at a time, and this overwrites the previously stored copy. This method is similar to the note pad, but has no intermediary store for information, just a simple linear buffer.

Interfaced files.

This method allows users to create an intermediary file full of information, which can then be processed by another application. This method is useful when a large amount of information has to be exchanged between two applications.

Integration and consistency are closely linked, with consistent systems being easier to integrate. As with consistency, integration between systems from the same software house should be attainable (although technically challenging), while integration with products from other software houses is again more difficult.

2.4.1. Modes.

The concept of modes is considered by many an antithesis to integration [Tesler, L:1981]. It is suggested that modes cause both novice and expert users considerable problems [Sneeringer, J:1978]. The idea of a mode has developed from the hierarchical structure of computer systems; that is, systems comprised of sub-systems made up of smaller sub-systems, and so on. Associated with each sub-system is a set of commands or operations,

which can be applied to a second set of objects. Although these sets often overlap, there is the problem of remembering which operations and objects apply to each sub-system or mode. This problem is also compounded by the user having to remember which mode they are in, and how they reached that point [Swinehart, D.C:1974]. There is also the potential problem of identical operations having different effects in different modes. A final problem is the freedom to transfer information easily from one mode to another.

The counter argument is that modes are both 'natural' and beneficial when using software systems [Canter, D:1985]. In examining the real world there are many instances of behavioural modes. For example, dining at an expensive restaurant as opposed to eating at a snack bar. Situations and environments often dictate certain types of behaviour from individuals placed in that environment, and these conditions provide security by establishing boundaries of behaviour agreed by society. At the application level, modes are unavoidable, with different applications for Word Processing, Databases, Graphics Editors, Desk Top Publishing, and other specialised software. However, the undesirable effects of modes described above need addressing, and may be reduced by the use of other techniques. Modes can be used to provide contexts in which a user can work. By making clear which operations and objects are applicable in each context and what their meanings are, the user can have the security of knowing whereabouts they are in a system, and what they can do [Carter, J.A:1987].

Provided the modes, operations, and objects associated with each application are explicitly defined, an interface can give mode sensitive advice to users. This alleviates the need to memorise sets of operations. An interface could also provide information concerning the current position of a user within a system. That is, which mode they are in and how they got there. This information could be presented using graphics or a simple text description, and should alleviate the problem of 'getting lost' within the functionality of a system. The problem of identical operations having different effects in different modes, is due to inconsistency between sub-systems.

Finally, an application and its interface could allow users to open, or use, more than one mode at a time. They could then copy information from one mode into another. Obvious constraints involve graphical and text information, where conversion is difficult. One tried and tested solution is to use windows, coupled with the 'What You See Is What You Get'

(WYSIWYG) concept [Shneiderman, B:1987]. Each window represents a separate application or mode, with its own set of operations and objects. Although user interaction may only occur in one window at a time, many windows may be active and multi-tasking supported. The integration techniques described earlier can then be used to enable information transfer between different windows.

2.5. Metaphor.

Essentially, a Metaphor provides a link between real world concepts and ideas, and equivalent computer concepts and ideas. It enables users, both novice and expert, to learn quickly and adjust to new computer applications, thus making them easier to use. One example is that of the Desk Top, used with several operating systems, including Apple Macintosh and the Graphics Environment Manager (GEM) developed by Digital Research. Here, graphical icons are used to represent various computer hardware and software components - disks are represented using a Filing Cabinet icon, and directories using a Folder icon. Individual files each have an icon representing their purpose, while files and directories are deleted by dragging an icon and putting it in a Waste Paper Basket represented by another icon.

A Metaphor is an important feature of user interfaces [Drake, K:1985]. Often, the virtue of an interface does not lie in the efficiency of its Metaphors, but in their familiarity [Edwards, S:1983]. For example, the hand calculator Metaphor where a picture of a calculator appears on the screen. If available, a hand held calculator is much faster to use, but the Metaphor is familiar to most users.

The icon is one of the most common Metaphors used in new interfaces. An icon is simply a small picture used to convey an idea, or other information, within an interface. Underlying this is the assumption that we live in a strongly visual and spatially organised environment [Rogers, Y:1989]. Icons provide a pictorial representation for various aspects of a Metaphor. There are many different types of icons, and their usefulness is the subject of current research [Benest, I.D:1987], [Card, S.K:1987], [Glinert, E.P:1987].

A Metaphor potentially offers many advantages to the interface designer, and in many respects is both natural and unavoidable within the computer interface. A Metaphor can portray complicated universal ideas, using simple

pictures, and concepts, avoiding the need for lengthy verbose descriptions. However, this powerful tool must be used carefully, as various problems exist. These are now discussed.

The understanding of a Metaphor is often related to cultural background, as different countries have their own languages, protocols, and concepts. A Metaphor must be chosen to avoid these differences, using commonly understood ideas such as filing cabinets, and folders. A Metaphor must also be obvious and easy to grasp. If users have to look up meanings of particular Metaphors, then their use is pointless.

Metaphors, and particularly icons, can be context sensitive, which may cause certain inconsistencies. For example, a magnifying glass may be used to represent a button for selecting a more detailed view of a picture. However, the same magnifying glass may be used to represent a function for examining the text within a file which sits in a particular directory.

In an effort to imitate the familiar object, one may also imitate the limitations of an object [Houston, T:1983]. A Word Processor patterned after a typewriter so slavishly that you see a graphical image of the type ball swing up each time you type a character may be comfortably familiar, however it can also become a distraction.

The interpretation of a Metaphor depends upon the knowledge associations, and previous experience of the user [Abrams, K.H:1987]. For example, an icon which has a picture of a specialised surgical tool, will probably only be understood by medical people. When using Metaphors, the interface designer must therefore carefully consider the background, and expertise of the user group before deciding which ideas to represent within the interface.

Medical and Psychological theory suggests that the use of Metaphors allows people to make use of more of the thought centres of the brain [Benzon, B:1985], [Pope, A:1983]. This is an improvement over conventional interfaces, which are thought to only stimulate the logical, analytical, and calculatory centres. However, it is improbable that this is a major motivation in development of new Direct Manipulation interface technology which is predominantly rich in Metaphors.

Finally, Metaphors should be part of the interface, separate from the application, and made as explicit as possible. Following these guide-lines, it

should be possible to effectively modify the Metaphors of an interface according to the characteristics of different user groups.

2.6. Interaction Styles.

Interaction Style governs the overall method of interaction with a computer system, and affects what the final interface looks like. There are three main divisions with many hybrids, and these are described in the following sub-sections. Each of these interaction styles has its own advantages, disadvantages and its own dedicated group of supporters in the field of Human Computer Interaction. Certain styles are more suited to certain applications, and are preferred by certain types of users, who will often change style as their expertise increases.

The interaction style preferred by a user is very much a subjective choice, and it is therefore difficult to design algorithms which accurately match different styles to individual users. It is more important that an interface offers a choice of interaction styles to the user. It may also be useful to allow users to see their interaction with one particular style interpreted and executed in another style. For example, the selection of a menu item which not only selects the relevant function, but also displays the equivalent command in another part of the interface. This may encourage users to move between different styles, and select an appropriate one for their task. It again requires that the application and interface be clearly separated, and should result in an application functionality which is style independent, whilst the user interface itself can support different styles.

2.6.1. Command and Natural Language.

This style of interaction involves presenting the user with a prompt (e.g. a flashing cursor), and expecting them to express their requirements in Command or Natural Language. Users have to translate from their perception of intent to a grammatical syntax which the computer understands. Command language is a sub-set of Natural Language, designed for computer efficiency and processing speed [Benbasat, I:1984]. In many situations Natural Language is the best form of input, as the user can specify their requirements in English (or other natural language) without having to learn the system dictated command structure. However, because of the great complexity of language, Natural Language interfaces are difficult to design [Grace, J.E:1987].

Several problems arise with Command Languages. Users may not know what commands are available, and therefore find it difficult to remember syntax and names [Minor, S:1987]. Users moving from one system to another may also have problems with different names for similar functions, e.g. DIR in IBM DOS and LS in UNIX, which both list directory contents. Command abbreviations may be useful, but the type of abbreviation may be difficult to choose [Jones, J:1988]. Generally the problem areas arise from having to memorise command names, syntax, and their associated functions. However, one advantage of Command Language is the possible speed and expressive power when used frequently by expert users.

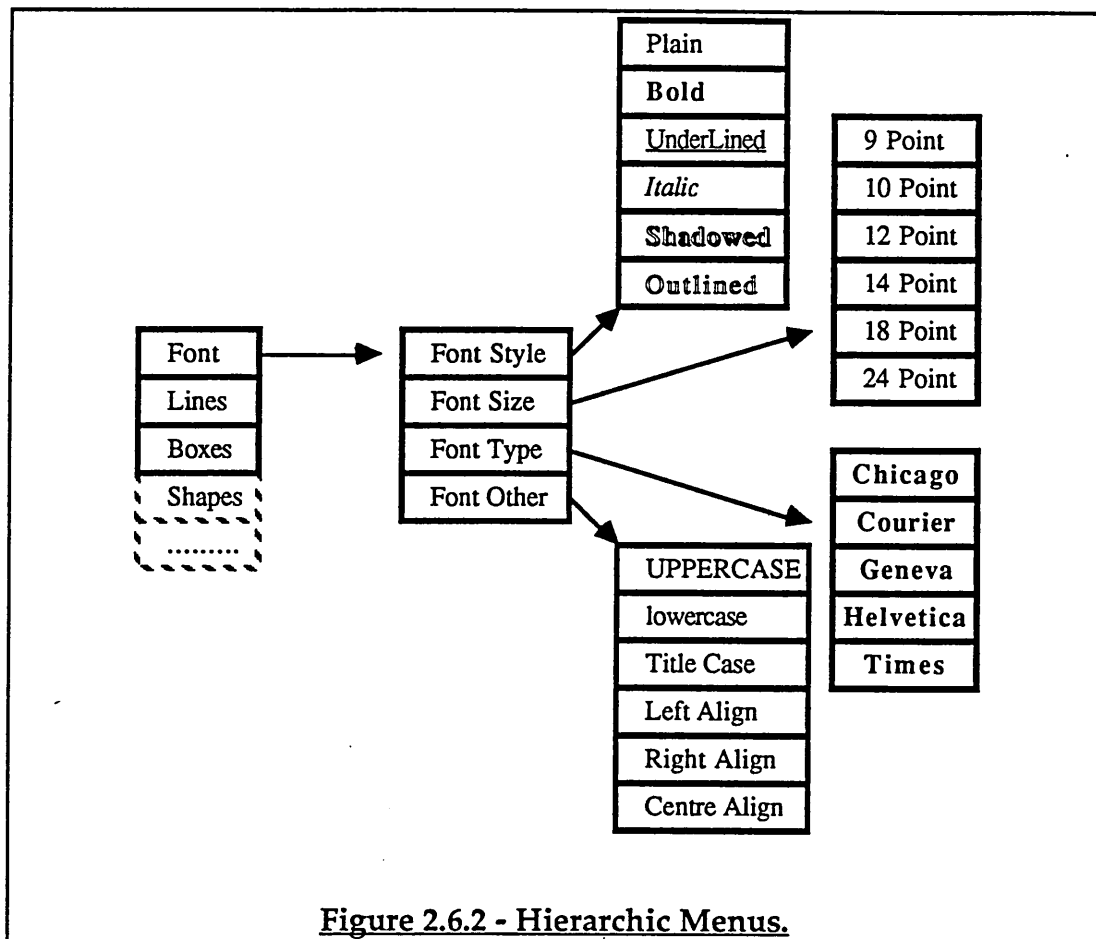
Natural Language is potentially better than Command Language, being far more flexible and adaptable. Users do not have to remember difficult names and syntax, as Natural Language interfaces should be able to recognise alternative names and syntax [Abrams, K.H:1987]. Novice users should find learning to use Natural Language interfaces easier and quicker. Also, as they become more expert, they can derive their own Command Language sub-set which suits them individually. However, the naturalness of such interfaces is questioned [Dillon, A:1988], [Ogden, C:1987]. It is natural to speak, and hear, language, but is it really so natural to type in this dialogue at a computer keyboard ? Natural Language may be better suited to multi-media input/output, where voice recognition and generation hardware is available [Multiworks:1989], [MultiPoint:1989], [SOMIW:1989].

One comment relating to both Command and Natural Language systems concerns the dialogue structure. The user is expected to type in a command, the computer then processes this command and returns a suitable response. When the user makes a mistake, this dialogue could become difficult to use in error correction and advice consultation. It may also become tedious, and there is the problem of waiting for system responses. The computer does not always show what it is doing, and users can be left waiting (sometimes anxiously) when they press the return key at the end of a command.

Natural Language processing is a wide field of interest to both the Artificial Intelligence and Human Computer Interaction research community. Many more questions, problems, and solutions exist which are outside the scope of this research. These concerns are addressed within the relevant research disciplines [Barlow, J:1989], [Bench-Capon, T.J.M:1989], [McKoy, K.F:1988], [Meyer, B:1985], [Schroder, M:1988].

2.6.2. Menus.

A Menu driven system, presents the user with a list of options from which they can choose. One option may lead to another menu being displayed, and options from there may do likewise. As figure 2.6.2 shows, the result is an inverted tree, with nodes within the tree (that is nodes to which downward branches are attached) being menus, and end nodes being the final result of an interaction.



The main advantage of this style is that all the available system functions are displayed. The user does not have to remember complicated syntax, and only needs to know how to interpret the menus and make their selection. Such a style is therefore useful for novice users.

However, when the number of menus is great and the depth of the tree (i.e. number of connected menus) is large, problems arise [Brown, J.W:1982]. One such problem is selection speed. When a user has to navigate through numerous connected menus, it takes a considerable length of time, which is unacceptable to regular expert users. The interaction process: read list and

select choice - can again become tedious. Also, correct functionality often depends upon well structured and well worded menus.

Another problem with menus is that of weak associations between choice descriptions, and final selections [Snowberry, K:1985]. In a system where there are many connected menus, the user may have to pass through a large number of menus before arriving at their required function. At each of these menus the user may get lost, and go down the wrong branch of the tree by making the wrong choice. The choice descriptions must be carefully chosen so that they correctly describe the sub-menus which follow from that choice. The more sub-menus, the weaker the association between a choice description and the final function. Therefore, it is more difficult for a user to select the required function, and there is a greater possibility of 'getting lost'.

Menus serve two main purposes, chiefly selecting functions, and representing a particular state from a list of possible states [Erklundh, K.S:1987]. For example, a selected font from an available list of fonts. In either case the menu structure can be separated from how the menu is actually presented and how it is used. In some systems, menus are invoked by pressing a certain mouse button while pointing to a particular window, and in others by moving a cursor over a particular word representing the title of the menu. In some systems hierarchical menus are also used, whereby menu items may lead to other menus. In other systems, certain groups of items are separated using lines or boxes. The actual menu structure can be described using a formal syntax which is separate from how the menu is actually presented, and used. The presentation and usage of a menu ultimately depends upon the interface style, whereas the underlying menu description remains constant. One well defined formal menu description grammar is known as 'Lean Cuisine' [Apperley, M.D:1989]. The potential for this form of separation is discussed in section 6.4.14, where an extended 'Lean Cuisine' implementation is presented.

2.6.3. Direct Manipulation.

This is the latest advance in Human Computer Interaction, and is heralded by many as the interaction style for the future [Shneiderman, B:1983]. Direct Manipulation systems involve a considerable amount of user interaction and Metaphors, with users moving objects (represented as pictures or icons) around the screen, selecting tasks and objects to be worked on. Such systems

include the Apple Macintosh family of software, and the Graphical Environment Management (GEM) developed by Digital Research.

Direct Manipulation is based on the principle of "What You See Is What You Get" [Warfield, R.W:1983]. Users are expected to, and usually do, grasp basic conceptual ideas. These ideas relate to the selection and manipulation of objects and tasks, and what happens as a result of their selection and manipulation.

There are many examples of successful Direct Manipulation systems [Lesniewski, A:1987], [Quint, V:1987], and experiment has shown them to be readily accepted by novice and expert users [Shneiderman, B:1987]. Novice users usually find learning and knowledge retention easy, and quickly grasp the basic concepts of such systems. Meanwhile, expert users find complex problems quick and easy to express. However there are some problems, mainly due to inconsistencies, and the use of poor Metaphors.

The problems with Direct Manipulation stem directly from the very features which provide its strength. Direct Manipulation is based upon a small but powerful set of interactions which are rich with Metaphor - for example dragging and selecting icons using mouse movement and button presses. These interactions are easily remembered by users, who expect similar effects for the same interactions in different applications. When the effects are inconsistent, cognitive problems result from having to temporarily unlearn previously well understood concepts. Similar problems occur when particular Metaphors are misrepresented, or misunderstood.

2.6.4. Combinations.

Many interfaces combine the above styles to form hybrids. Direct Manipulation and Menu interaction are often combined, as they use similar Metaphors and user interactions. However, Command and Natural Language styles are rarely combined with Menus or Direct Manipulation because of the differences between them.

One hybrid is Form Filling [Frohlich, D.M:1985], where the user is presented with a screen full of questions to answer in order to complete a certain task. The form may be displayed as a result of typing in a command, or selecting a function from a menu. The response to these questions may be typed, or

selected from menu lists. The complete form then provides the basis for executing a particular task, or sequence of tasks.

2.6.5. Dialogue Control and Specification.

Underlying all interface styles is a specific dialogue control mechanism. This describes the permissible user interactions, and the resulting effects upon the interface and application functionality. As dialogue control is unavoidable, it is preferable to make dialogue control mechanisms explicit within the interface. This may take many forms and could serve several purposes. Current descriptions include Task Action Grammar [Payne, S.J:1986], Command Language Grammar [Moran, T.P:1981], and 'Goals Operations Methods and Selections' (GOMS) [Card, S.K:1983].

Explicit dialogue descriptions should be executable, which would enable novel styles to be easily tested, and encourage interface designers to explore new interaction techniques [Alexander, H:1987], [Jeremaes, P:1987]. Libraries of dialogue styles can also be accumulated and easily modified. Finally, interface consistency should be improved by automatic style analysis and cross referencing.

The use of executable dialogue descriptions again requires separation between the interface and application functionality. Styles can then be ported between different applications, which should improve the consistency between them.

2.6.6. Style Guides.

Interface style is implementation independent, as it is possible to define various interface styles, and implement them using different tools or languages. Interface styles should therefore take the written form of style guides. These guides can then be used as reference works by interface designers, to implement consistent interfaces for disparate applications. In addition, executable style descriptions may also be attached to these style guides, for use within an interface design Tool-set.

These style guides may be marketed commercially, and distributed to different software houses. This should encourage interface standards, and several style guides are already available including Motif [Oldenburg, H:1989] and 'Touch and Feel' [Kluger, L:1989]. However, the written style guide

approach may also lead to profiteering by the combined enforcement of standards, and use of copyright controls.

2.7. Error Handling.

Software should be designed to deal with the mistakes which users inevitably make. The prime objective of error handling must always be to assist learning, and correct misunderstanding. Most applications merely detect those user actions which would cause system problems, generate an error message, and display it. This is useful, especially when the application is simple and the error messages are easy to understand. However, new applications must provide better facilities. As described below, error classification is application independent and the same classification method can be used for different applications. It is therefore contended that the responsibility for error handling should lie within the user interface, which must at least include advice giving components which can suggest the reasons for an error, and provide relevant advice/documentation. This approach leads the design of generic Intelligent Help System and Error Handling modules incorporated as part of the user interface (see section 3.2.1).

Every application has a functional structure, which formally defines correct and incorrect usage. Application structure may be explicitly defined using various Systems Analysis and Design tools, or may exist only in the head of the designer. Application functional structure is presented through a user interface, therefore the Application Model is a product of both application functionality, and user interface presentation. Similarly, a user has a particular model or understanding of how they expect an application to work - that is how to instantiate tasks, what response to expect, and the understanding of Metaphorical meaning. Errors occur as a result of mismatches between these two separate models. Although the Application Model is usually fixed, the User Model is often incomplete, unstable, and confused [Norman, D.A:1983]. This inconsistency must be dealt with by suitable error handling mechanisms.

The type of mismatch can be classified according to various frameworks. One such classification is Evaluative Classification Methodology [Booth, P:1987]. The following definitions are used, and figure 2.7 shows the classification system.

Objects.

An object is, in essence a thing to which something is done, or about which something acts or operates. For example a data file in an application, a figure, or a character. An object mismatch can take the form of one of three possible types; an object-concept mismatch, an object-symbol mismatch, or an object-context mismatch. An object mismatch might be said to have occurred where an object is unfamiliar to a user or has unexpected and unwanted properties (concept mismatch), an object is misrepresented (symbol mismatch) or an object may appear in the wrong mode or situation as far as the user is concerned (context mismatch).

Operations.

An operation is an action which is performed upon an object or objects within the application - for example, saving a file, deleting a character, changing a shape in a graphics package. Again, there are three types of operation mismatch. An operation-concept mismatch is where the application cannot perform the operation in the way that the user intends. An operational-symbol mismatch is where an operation is misnamed as far as the user is concerned, and an operation-context mismatch is where an operation cannot be performed in the way that a user would like, in a particular situation or mode.

Concept.

A concept may be either an object or operation whether represented mentally (in the user) or in lines of code (the computer). A concept mismatch is a fundamental difference in the understanding and representation of application objects or operations.

Symbol.

This is taken to mean a word, character, sign, figure, shape, or icon employed by either the user or the application to represent an object or operation within the application. A symbol mismatch is not one of fundamental understanding, but occurs where the application and the user adopt different terms to represent the same concept.

Context.

This is the arrangement of, and the relations between, the objects within an application at any point in time. An object-context mismatch is then, where an object is in the wrong situation or position as far as the user is concerned. An operation-context mismatch is where an operation that can be performed in other circumstances either cannot be performed in the way the user intends, or cannot be performed at all.

		Object	Operation
Context	Concept	Object-Concept Mismatch	Operation-Concept Mismatch
	Symbol	Object-Symbol Mismatch	Operation-Symbol Mismatch
	Concept	Object-Concept Context Mismatch	Operation-Concept Context Mismatch
	Symbol	Object-Symbol Context Mismatch	Operation-Symbol Context Mismatch

Figure 2.7 - Error Classification Methodology.

The majority of errors can be fitted into this classification framework, apart from inefficiency errors [Elkerton, J:1987]. In the latter case, the user understands an application and how it works, but does not realise that the application provides a more efficient mechanism for achieving a certain goal. For instance, consider an example which is concerned with deleting multiple files - a user may use individual delete commands for each file, while being unaware of a multiple delete command, which takes a list of files to delete as an argument. Such inefficiencies waste time, and should be prevented whenever possible.

Having established an Error Classification Methodology, a means of recognising or detecting errors is required. Most errors cause a failure in the dialogue between the user and the interface - for example, incorrect commands, or trying to select invalid or inoperative functions from a menu. Such errors are easily detected, as the application or interface is designed to expect only certain types of response. The inefficient use of application functions is more difficult to detect. Artificial Intelligence is needed to monitor the interactions performed by a user, predict the intended

goal, and suggest an alternative sequence of interactions which achieves the same goal more efficiently. Knowledge is therefore required concerning application functionality and permissible user interactions.

Different interaction styles present, and restrict, certain types of errors. Command and Natural Language styles allow a greater range of errors to be made, and provide a mechanism for detecting the different types. Because user interaction takes the form of worded command sequences, these can be analysed to see whether particular types of mismatch occur [Smith, J.J:1985]. Direct Manipulation and Menu interaction styles present different problems. Typically, the user is prevented from performing incorrect tasks. Users are also prevented from using invalid symbols and operations as they are only allowed to choose from the symbols and operations presented to them. Some examples are listed :-

- the use of Menus which only list valid tasks, or operations
- invalid or inoperative tasks in a menu list are usually presented in a different type-face
- pointer movement is often restricted, depending upon what task is being performed, or which mode an application is in.

Because of the restricted interaction, mismatches are more difficult to detect, and this can be misleading. It is wrong to imply that because a user is prevented from performing certain operations, or using certain objects, that they understand why this is so, as a user may wish to invoke an invalid operation or incorrectly use an object. This type of error could provide an interface with useful information concerning certain mismatches between the User Model and Application Model. Questions arise as to whether these restrictions should be removed. Instead, errors could be detected and corrected using Intelligent Help and Error Handling systems. Such concerns are outside the bounds of this research.

Another pertinent issue is that of error description and advice giving. Most applications usually display a simple error message, with the option of a more detailed description. This is insufficient for complex applications, where more assistance is required to help identify the cause of an error, and to offer advice on how to avoid it in the future. Current research shows that Expert System technology is necessary to meet these requirements [Carroll, J.M:1987].

Errors can have different effects on different users, ranging from sheer panic to disregard. It is therefore important that an interface provides a 'forgiving' environment, reassuring the user that their mistakes are not critical, and that any abnormal effects can be easily undone.

All errors must be resolved at their occurrence before the system proceeds. It is important that the user is informed as rapidly as possible when an error is detected, and suitable attention attraction mechanisms should be used, e.g. Bell, Flashing Red Colour. However, this mechanism must not be so intrusive as to inform non-users near the system of the mistakes made by another user [Thimbleby, H:1986]. When an error is detected, users should be returned to a system state which existed before the last command was issued which caused the error. This provides a 'forgiving environment' and encourages users to explore the system.

Error messages should be positive. For example, the message 'Deletion not allowed BECAUSE you do not have sufficient security access' is less provocative than 'access not allowed'. An error message should not deter the user from exploring the application. It should reassure the user that no unpredictable side-effects have resulted from the error. The user may then continue confidently, knowing that the system is in a predictable state.

The issues being addressed within the field of error handling and advice giving are complex, needing expertise from both Artificial Intelligence and Expert System technology. Chapter three examines the role of Intelligent Help Systems as a means of providing better error handling and advice giving facilities. This chapter also presents an overview of current research within this field.

2.8. Documentation and Tutorials.

The success or failure of a software application can often be influenced by the quality of the accompanying documentation. The process of producing documentation can sometimes be a difficult and lengthy task. Listed below are some basic guide-lines which draw attention to the various concerns within this field.

Documentation may take one of two media forms: the printed page, and on-line documentation. The printed page is most common, with various formats ranging from single page command summaries to extensive, fully

indexed, detailed manuals. Alternatively, on-line documentation uses computer media, where the text and pictures are stored in a form which can be read and presented to the user through an interface.

Documentation should use consistent terms, e.g. when using the term 'screen layout' it should not suddenly change to 'display format' at a later date. It should also have consistent structure across different software applications, e.g. Introduction, Index, Chapters, Command and Feature summary, and Appendices.

Documentation should be simple to understand and clearly laid out, using as many examples as are feasible. It should cater for both novice and expert users, including special sections for expert users and concise, readable command and function summaries.

Documentation should not assume that users have expert computer knowledge, and any expectations should be clearly laid out, i.e. previous experience, or manuals previously read [Wendel, R:1987]. There may also be a need for manuals, tutorials, and help facilities on the subject of computer skills, e.g. using disks, keyboard, mouse, printer, files, and directories.

Documentation should encourage users to get hands on experience as soon as possible. Most users are keen to sit at terminals and start using the system as soon as possible [Carroll, J.M:1983]. Long tedious documentation has a dysfunctional effect on people, and inhibits the learning process. The result of this is counter-productive, as users typically try and use the system out of desperation with no understanding or guidance. This inevitably results in frustration and causes discouraging system errors.

Good programs do not need extensive comments, as the program code itself should be clear and easily understood. Similarly, applications should not require lengthy documentation to explain how to use them, as the functional model should also be clear and easily understood. On-line documentation for a particular system enables users to quickly access information to assist them with their problem. This documentation could also be linked to error messages. The only drawback with this facility is the possible loss of software protection. With present software, lengthy manuals are often required to use the software. Although the software can often be easily illegally copied, documentation is more difficult and costly to duplicate, thus restricting improper use of the software.

When on-line documentation is provided, context can also be used to improve its accessibility [Moll, T:1987]. This can assist the user in finding the relevant help or documentation. The interface effectively provides assistance based upon where the user is in the system, what the user is doing, and what the user has recently done. Ultimately this facility also provides a type of 'panic button', to cope with the situation when the user is completely lost, or 'functionally disoriented'.

Interactive tutorials may also provide a useful teaching tool. These could take the form of real examples or problems with worked solutions, which are seen running within the application. Along the way, a tutorial shows the commands that must be entered and what they do within the framework of the specific example. Effectively, the user can watch the interface simulate a user using a particular application to solve a specific problem. One manifestation of this is the 'rolling demonstration', often used in sales presentations.

It should always be made clear to the user how help can be obtained, [Herbach, M: 1983]. The selection action should also be simple, e.g. pressing the F1 key. Essential help which is often required by novice (and sometimes expert) users is orientation and familiarisation. That is, what keys do what, and what part of the screen holds what information. This basic information should always be made clear to novice users at an early stage, as this aids the learning process.

One useful addition to conventional on-line help techniques would be to allow users to add their own comments to an existing documentation knowledge base. Having requested, and received the appropriate help, users could then have the option of adding their own comments to the help displayed.

On-line documentation and interactive tutorials are closely related to error handling. Again, Artificial Intelligence and Expert System technologies have a great deal to contribute to this field. Errors often mark the start of a tutorial or advice giving session, therefore knowledge needs to be shared between the error handling and advice giving components of an interface.

2.9. Interface Separation.

Interface separation is a growing concern. If user interfaces were completely inseparable from their application, there would be no such thing as an alternative, customisable, or adaptive interface. Some degree of separation is already possible, because various interface Tool-sets are currently available to the interface designer, for example Open Dialogue [Patel, H:1989], HyperNews [Pearce, D:1989], and XWindows [Sun:1990]. As a result, the following questions arise :-

- what is the scope and definition of the application ?
- what is the scope and definition of the user interface ?
- where can the separation line be drawn, and how ?
- what are the advantages of separation ?
- what are the effects and limitation of separation ? Especially :-
- what constraints does the application put on the separable user interface ?
- what constraints does the user interface put on the separable application ?

Using terminology suggested by Cockton [Cockton, G:1987], it is proposed that a software application can be divided into a Non-Interactive Core of application functions and a separate user interface, which can communicate with each other. The Non-Interactive Core of application functions does not specify any User Interaction dialogue control. Effectively, it cannot directly request user input, or generate any screen output. These tasks remain within the user interface, which conversely does not implement any application functions. Messages are sent to the application to instantiate functions, and inquire on states. Similarly, results are returned by the application to the user interface using arguments, or tokens. This link between application functions and user interface should be formally defined.

The potential advantages of complete user interface separation are numerous :-

- consistent user interfaces should result from the re-use of separate interface components within different interfaces
- the same application may have many different interfaces which can be easily tailored and adapted to individual users or organisations

- an interface can easily change over a period of time to meet the changing needs of individual users
- new Human Computer Interaction technology can be utilised when it becomes available, and the interface updated accordingly without affecting the application functions
- prototyping of interfaces can be performed separate from the development and implementation of the application functions
- cost and time savings arise from many features, including prototyping, customisation for individuals, and interface maintainability
- application can be designed without Input / Output or Human Computer Interaction considerations.

These advantages however, depend upon the extent of separation supported within the software architecture.

A truly generic interface is impossible to achieve. This is theoretically defined as a user interface which can be used for many functionally different applications, and implies that a Word Processor style interface can be used to interact with a Database, or Graphics application. This would cause the user many cognitive problems, as the underlying interface metaphor would be difficult to match to application functions. Application functions therefore do affect and constrain the user interface. Likewise, the interface constrains the application functions. These constraints are not necessarily adverse, and require further investigation.

This research focuses upon interface separation. A new interface software architecture is proposed and implemented using an object oriented language. This architecture uses separation, and is later evaluated in order to derive the benefits and constraints that result from separation, and answer the questions posed above.

2.10. Interface Ergonomics.

The primary objective of Ergonomics is the matching of a job to a particular worker, with the aim of reducing the workload upon them [Matilla, M:1987]. It is a well founded research area with many applications in the industrial and commercial sectors. Interface Ergonomics has a primary objective to match a software application to a particular user, again with the aim of reducing the workload upon them. In the case of Interface Ergonomics this

workload takes a mental form and is typically cognitive in nature, as opposed to Ergonomics where the workload is usually physical.

Interface Ergonomics addresses many questions which relate to user interface software [Balzert, H:1987]. These include :-

Use of highlighted and flashing text.

Screen layout.

That is where particular text and graphic items such as error messages and help information should be positioned.

Metaphor.

That is what ideas to represent and what metaphor to use.

Dialogue style.

This should be matched to individual users, and the type of task that they are trying to perform.

Undoing.

Actions should be discrete and their effects reversible. Users should be offered an 'Undo' function, and the extent of this, i.e. the number of sequential commands which can be undone, should be adjustable [Waldor, K:1987].

Over-typing.

Consideration should be given to over-typing. The effect of over-typing can be accomplished by combination of delete and insert actions. By preventing over-typing, users need not worry about whether they are in insert or over-type mode.

Auto Repeat keys.

Work by Thimbleby [Thimbleby, H. 1986] suggests that auto repeat keys should be avoided (that is keys repeating themselves when held down,). He suggests that a separate repeat key should be provided, and not allowed when using 'dangerous' keys. Another alternative is to provide key clicks which provide a positive response to the user that a key was pressed.

Response Time

The response time of an interface is important. People are used to dealing with real world response times in the order of several seconds at the longest, e.g. turning a steering wheel, changing channel on a T.V. set. Although in certain conditions people accept response times in the order of several seconds, e.g. phoning, there is an intermediary response, i.e. a dialling tone. This intermediary response reassures the person that their request is currently being dealt with and that they must wait. It also gives them the choice of cancelling an action they know is currently being processed, returning the system to its original state before the action was initiated. Other circumstances where large response times are acceptable include starting or initialising a new system, and switching off or closing down a system, e.g. turning on a T.V. set. Long response delays reduce confidence in the system and disrupt work flow.

User interfaces should be aware of this, so that any delay between successive application states never exceed a maximum value for example, a maximum response time of 2 seconds [Thimbleby, H:1986]. Where this is not possible, intermediary visual responses should be generated. If a user decides that they wish to cancel a task during this intermediary display, the system should then be returned to the previous state before the task was initiated.

Type Ahead.

Typing ahead is a useful feature for expert users, however, it can be frustrating for novices. Type ahead buffers should be variable in sizes and optional. Where typing ahead is allowed, the resulting intermediary displays should not be displayed, unless the response time again exceeds a pre-defined maximum value [Thimbleby, H:1986].

This area falls under the responsibility of Cognitive Psychologists whose skills lie in understanding how humans function cognitively [Dillon, A:1987]. Primary concerns involve the matching of various interface features and styles with appropriate human qualities [Rasmussen, J:1987].

The result of this type of work is the production of interface guide-lines. These can then be used to help the interface designer design better interfaces. For example, matching memory loads for the user by restricting the amount of information displayed on the screen.

Results from this field are usually accompanied by theoretical and empirical cognitive proofs. Although specific interface features can be identified, the correct setting for these features may be either a unique value or one of a finite set of values. For example, response time must always be less than a maximum of 2 seconds, whilst the colour of an error message could be either red, orange, or yellow. This encompasses the individuality of different users.

Most of the issues discussed in the previous sections are addressed by research within this field. It is the responsibility of the interface designer to take notice of Cognitive Psychology research results, and incorporate them into new interfaces. This should again be facilitated by the use of interface separation, and suitable interface design tools.

2.11. Summary.

This chapter has discussed the many software related influences which affect user acceptance of computer systems. The various affects of these influences were discussed, and improvements to existing technology suggested.

Several criteria were developed. When applied, these should assist the design and implementation of usable interfaces. The criteria are listed as follows :-

- separation between the interface and application
- explicit descriptions of interface components whenever possible
- the use of formal design methods
- development of new interface architectures and tools.

A need for new approaches to software design which incorporate these criteria was identified. These must focus attention upon designing software systems for the user, and also require suitable support tools and software architectures. It is contended that separation between interface and application functionality is paramount to any new design approach, and

must therefore be the basis for new software architectures and interface design tools.

Finally, the issues presented in this chapter are fundamental to any software application. They must therefore be correctly addressed before Artificial Intelligence is used to improve the interface. Experience has shown that failure to do so usually results in a powerful Intelligent Interface, which is even more difficult to use than its Non-Intelligent counterpart.

Chapter Three.

The Application of Artificial Intelligence to User Interface Design.

3.1. Introduction.

An Intelligent Interface is one which uses techniques drawn from the field of Artificial Intelligence to improve the user interface [Abrams, K.H:1987], [Carroll, J.M:1987], [Rivers, R:1989], [Self, J:1988]. This is achieved by adding extra modules which enable the interface to match certain user characteristics. Intelligent Interfaces also provide the facility to personalize an interface to suit an individual user. Knowledge is required about the individual user and their characteristics, and also about the variable interface features. Using this, an Intelligent Interface selects the correct feature values for a user, and consistently applies them across the entire interface.

Chapter two identified the main areas where software can be improved in respect to usability, while this chapter shows how some of these areas may be enhanced using Artificial Intelligence. Attention is given to interface architectures, and a modular Intelligent Interface design is proposed.

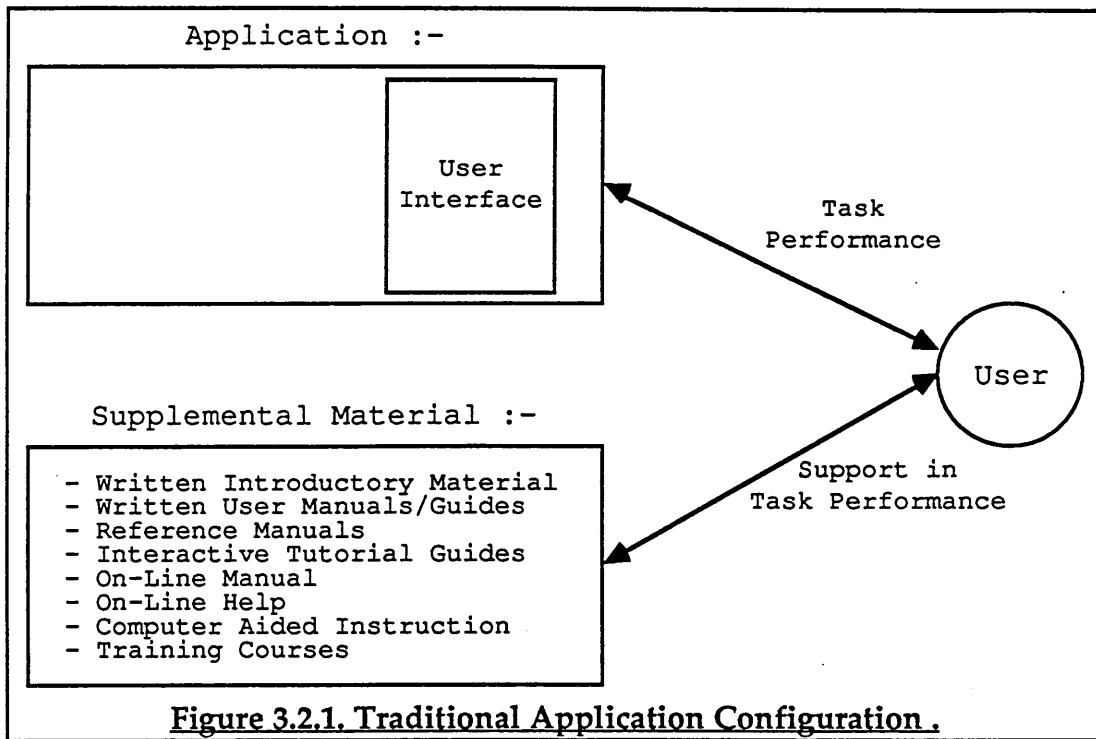
This chapter reviews current research into Intelligent Interfaces, and completes the background work for this thesis. Section 3.2 examines the major applications of Artificial Intelligence to the user interface. Section 3.3 presents a simple classification for interface architectures and design approaches. Section 3.4 examines the major design approaches, and lists the requirements which they must meet. This section also describes and evaluates several existing interface design approaches.

3.2. Additional Intelligent Interface Modules.

3.2.1. Intelligent Help Systems.

As discussed in chapter two and summarised in figure 3.2.1, traditional approaches to help systems are based upon separate application documentation, tutorial support, and training. Although documentation may be in a computerised on-line format, the initiative for satisfying application queries and for solving specific problems still remains with the user. The effectiveness of written or on-line documentation as a learning

tool is dependent upon the quality and clarity of the literature provided. Different styles of documentation are also necessary to assist the user as they progress from novice to expert. Unfortunately, documentation style is fixed and is usually aimed at the 'average' user. As a result, it can not take into consideration the different abilities and backgrounds of individual users who will use it.



As systems become more complex, training courses are also becoming a popular means of learning. Training courses are better suited to matching the learning abilities of different individuals, as professional tutors can tailor teaching material to personal competence. However, these courses tend to be expensive, and knowledge is easily forgotten over a period of time.

Another solution to user support is that of Computer Aided Instruction [Erlandsen, J:1987]. This typically entails a suite of programs which interactively teaches the user the salient features of the application. Computer Aided Instruction packages use the functionality of an application to provide a tutorial style approach to learning. The Computer Aided Instruction package effectively imitates a user, and illustrates on a step by step basis how to accomplish specific tasks, using the available application functions. However, Computer Aided Instruction packages have major shortcomings in that they are often introductory in nature. Again, these

packages usually assume an 'average' user, and therefore cannot be tailored to individual user abilities.

An alternate approach to user support is that of Intelligent Help Systems [Lutze, R:1987]. An Intelligent Help System provides on-line user support which is tailored to individual user requirements. These systems act as 'mechanised teachers' which monitor the progress of a user, and provide assistance which is adapted to suit their individual capability. For example, novice users are given verbose introductory help, while expert users are given shorter concise descriptions. An Intelligent Help System may also allow users to add their own comments to the existing documentation.

Intelligent Help Systems also need to incorporate Intelligent Error Handling techniques. Whenever a user makes an error, the Help System can leave the user to determine its cause and therefore correct it, or the system may assess the current capabilities of the user and intervene to offer a possible explanation for the error. Ultimately, it may also implement solutions upon behalf of the user.

An Intelligent Help System may be directly invoked by the user whenever assistance is required. It may also monitor the user, and when deemed necessary interrupt the user to offer what it determines as useful advice. For example, whenever a user is using the system inefficiently, or appears to be 'lost'.

Advice, or help sessions may take several forms, the simplest of which is a context sensitive description of relevant application features [Carter, J.A:1987]. Other styles include interactive tutorial demonstrations, and question and answer sessions. An interactive tutorial session provides the user with worked examples which demonstrate how a particular application function works. These examples may also be made more realistic by constructing them from examples taken from dialogue which has already taken place between the user and the application [Carroll, J.M:1987]. A question and answer session allows the user to interrogate interactively the help system in order to solve a particular problem [Hartley, J.R:1988]. The dialogue which occurs may also be used as feedback, to adjust intelligently the help system for future help sessions. In all cases, assistance is personalized to the individual capabilities of a user.

The main goal of an Intelligent Help System is therefore to support the construction of a correct and sufficient conceptual model about the application, by means of a behaviour similar to that of a human teacher [Miller, J.R:1987]. In doing so, account must be taken of the background, abilities, and previous interactions of the user. A balance must also be maintained concerning the initiative for providing assistance. A user may take the initiative to request help at any time. However, an Intelligent Help System may take the initiative itself at certain times, and interrupt user interaction to offer suitable advice. Unfortunately this may be dysfunctional, as it is also possible for these interruptions to disrupt user interaction, and therefore impede the progression of a user.

An Intelligent Help System can be clearly separated from the application. It requires knowledge concerning the application and its proper use, but this should be defined separate to the application itself. In order to work well, Intelligent Help Systems also require detailed knowledge concerning the available user interface dialogues and mechanisms, different teaching and explanation strategies, Natural Language, and user characteristics and traits. As described later in section 3.2.2.1, this knowledge is often implemented as computerised User Models, Application Models, and Real World Models. Generic Intelligent Help Systems can therefore be developed, and adapted to individual applications or users by changing the appropriate model. Various techniques are already available for representing these models in computerised format, and for determining their knowledge [Carroll, J.M:1988], [Sandberg,J:1988].

The major issues surrounding Intelligent Help Systems are related to the design and implementation of suitable computerised models [Murray, D.M:1987], the design of suitable intelligent inferencing mechanisms which monitor user interaction and modify the appropriate model knowledge [Kemke, C:1987], the selection and implementation of useful help session dialogue styles [Dix, A:1987], and the representation of human learning processes [Carroll, J.M:1987]. As a result, current work into Intelligent Help Systems benefits from research results derived within the disciplines of Artificial Intelligence, and Behavioural Psychology [Self, J:1988].

3.2.2. Models.

Models serve many purposes, and provide a powerful tool for the scientist [Williges, R:1987]. Essentially a model can be defined as any abstract representation of a real world phenomenon. This phenomenon may be physical or conceptual, and the nature of a model will depend upon its intrinsic aims. For the scientific community, models primarily enable knowledge concerning difficult real world problem domains to be represented in a clear, and understandable format. An example is Short Term Memory and Long Term Memory [Thomson, N:1986]. These two terms refer to an established Cognitive Psychology model which represents the way in which human memory functions. Scientific models can also be used as the basis for predictive reasoning in order to forecast events or situations which may occur in the future of the domain being represented. Finally, they may also be used as a basis for developing and testing new hypothesis, and in generating solutions to problems which arise from the domain of definition, and operation of the model.

A model is an abstraction and representation of some real world object or situation. Physical modelling is probably the most common form of modelling, and examples include sculptures, and pictures. Although useful, it is difficult to physically model metaphysical or conceptual ideas, for example the concept of how the human mind functions, or an individual's political motivations [Spall, R.P:1986]. In such cases, models are built around related facts, or hypotheses concerning the problem domain. These facts can be organised to form the basis of the knowledge contained in a model, and may appear as Natural Language statements, diagrams, mathematical values, or logic expressions.

The use of mathematics and logic as a modelling tool facilitates the generation of computable models. These models can be stored within a computer, and used as the basis for computerised inferencing or prediction. An Expert System is an example of a computable model and is comprised of separate knowledge (either mathematical values, or logic expressions) and heuristics which act on this knowledge in order to predict, or infer further knowledge [Kidd, A:1986]. The knowledge contained in a model can therefore be extended or modified manually by the Expert System user, or automatically by intelligent heuristic inferencing mechanisms.

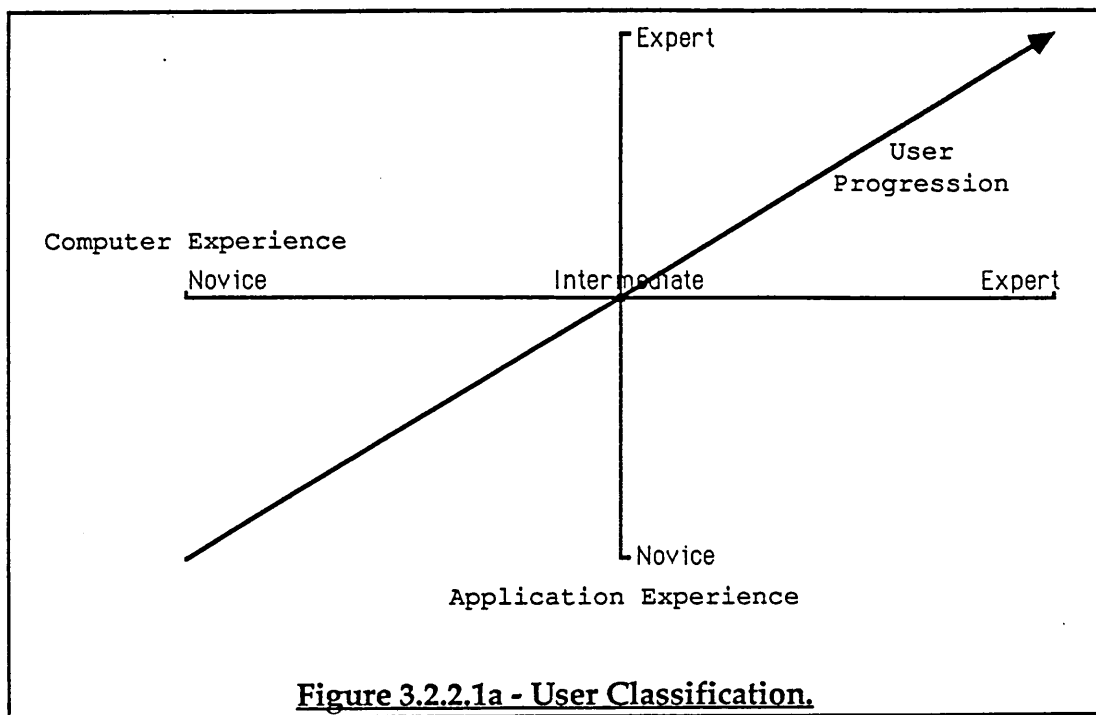
Models are not necessarily true representations, and may be hypothetical. They may also be difficult to prove or disprove, especially where models of human qualities are concerned. Models form a natural part of life, and provide a useful means of organising knowledge. Often, people do not realise that they are using implicit models to solve or understand problems. Every computer application and its interface assumes some form of model concerning the user and the real world.

As far as Human Computer Interaction is concerned, models serve two purposes. Firstly, models may be used as design tools to assist the interface designer in generating usable interfaces [Whitefield, A:1987]. Secondly, a model may be used by an actual interface in order to infer certain values, or characteristics, about a user, application, or the real world [Murray, D.M:1987]. The issues being addressed by Human Computer Interaction User Modelling research are related to the explicit definition of model structure, the application of models within the user interface, and the elicitation of knowledge which is contained within a model [Corbett, M:1987]. There are three types of model which are of particular interest to the Human Computer Interaction designer :-

- User Model
- Application Model
- Real World Model.

3.2.2.1. User Model.

To capture the difference between computer users, we need a classification framework to describe their individual characteristics. Users can be classified along two scales which are illustrated in figure 3.2.2.1a. One is their knowledge and understanding about the application domain, and the second is their knowledge and understanding about the computer and user interface domain. Both of these scales range from novice to expert, with the centre representing intermediate knowledge. A novice has no, or little knowledge or understanding, while an Expert has full knowledge and understanding. The position of a user on the scales varies relative to other users, and will also change with time, as they learn or forget knowledge.



These classification scales are useful, but problems arise with more complex applications. It is difficult to determine the range values, because a relative scale is required based on a definition of 'full knowledge' or understanding, and 'no knowledge' or lack of understanding. It is also difficult to define the granularity of the scaling, i.e. how many ideas and skills need to be learnt and understood before a novice progresses to become an expert user. Users also have other characteristics, such as previous experience with other applications, and cultural differences, which this does not represent. A richer classification, known as a User Model, is required.

A User Model, in respect to Human Computer Interaction, is a description of how users interact with software interfaces [Clowes, I:1985]. This model serves to describe the different activities which take place during the interaction between a user and a computer application. These activities include psychological, or mental thought processes, and physical actions such as key presses and mouse movement.

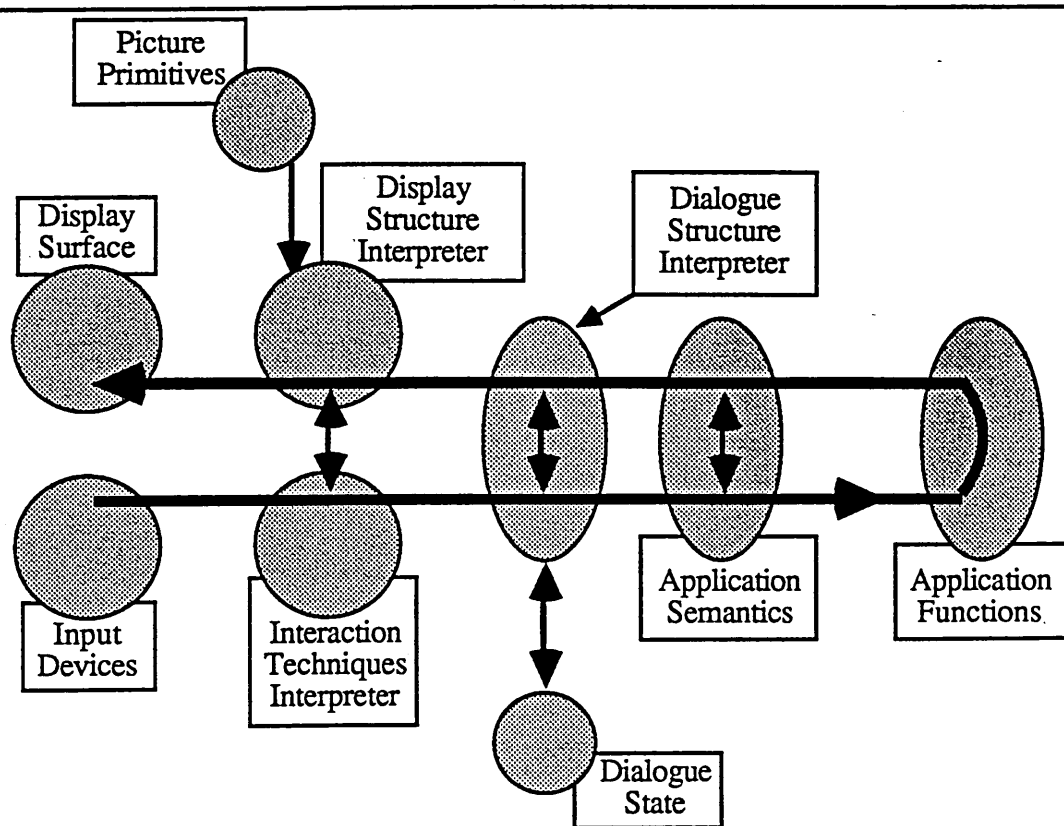
There are two broad types of User Model, namely: conceptual and quantitative models [Williges, R.C:1987]. Conceptual models are primarily concerned with representing cognitive processes, while quantitative models deal with the numerical representation of user performance.

Figure 3.2.2.1b shows two general models of human information processing. These originate from Norman [Norman, D.A:1986], and Wickens [Wickens,

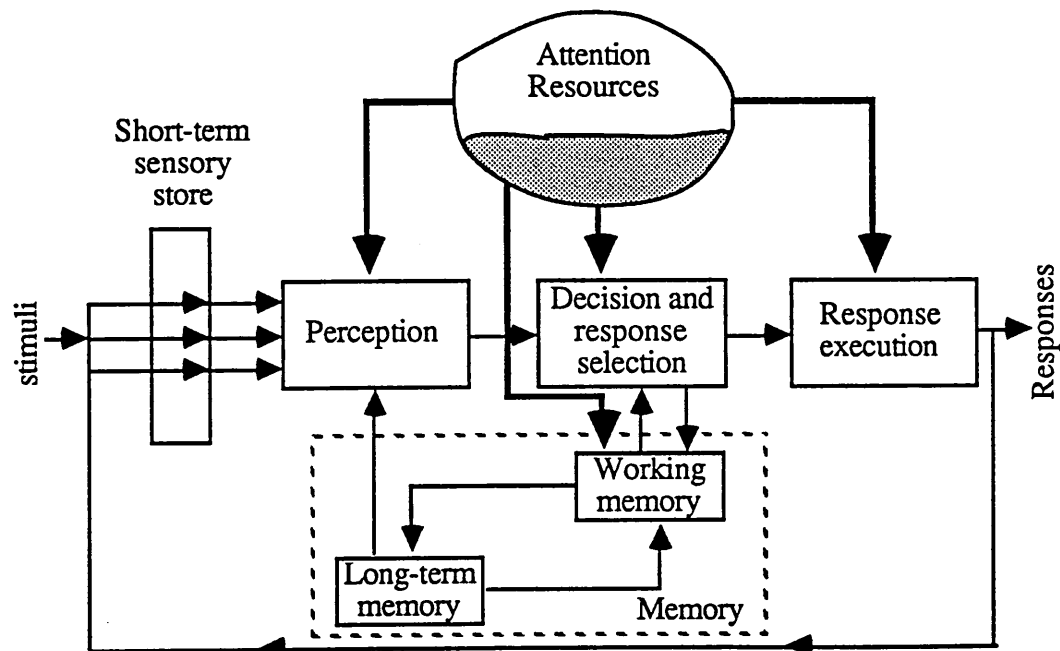
C.D:1984]. Conceptual Models serve to identify cognitive processes, cognitive structure, and cognitive strategy. Cognitive processes deal primarily with the procedural knowledge used by an individual while performing given tasks. Work by Norman [Norman, D.A:1986] summarises the cognitive strategy of a user, using 7 stages:-

- (1) Establishing Goal
- (2) Forming an Intention
- (3) Specifying the Action Sequence
- (4) Executing the Action
- (5) Perceiving the System State
- (6) Interpreting the State
- (7) Evaluating the system state with respect to goals and intentions.

Conceptual Models presented as tree diagrams, or networks, are often used to analyse the cognitive structure of knowledge used in a task. Kieras and Polson [Kieras, D:1985] make a distinction between two components of knowledge in operating a computer based system. First, there is the representation of tasks performed by the user, which can be stated as a hierarchical goal structure based on a production system composed of a collection of production rules. Secondly, there is the representation of the application which can be modelled by using transition networks consisting of a series of nodes connected by labelled arcs. As illustrated in figure 3.2.2.1c, these transition networks show the possible user actions and the possible resultant application states.

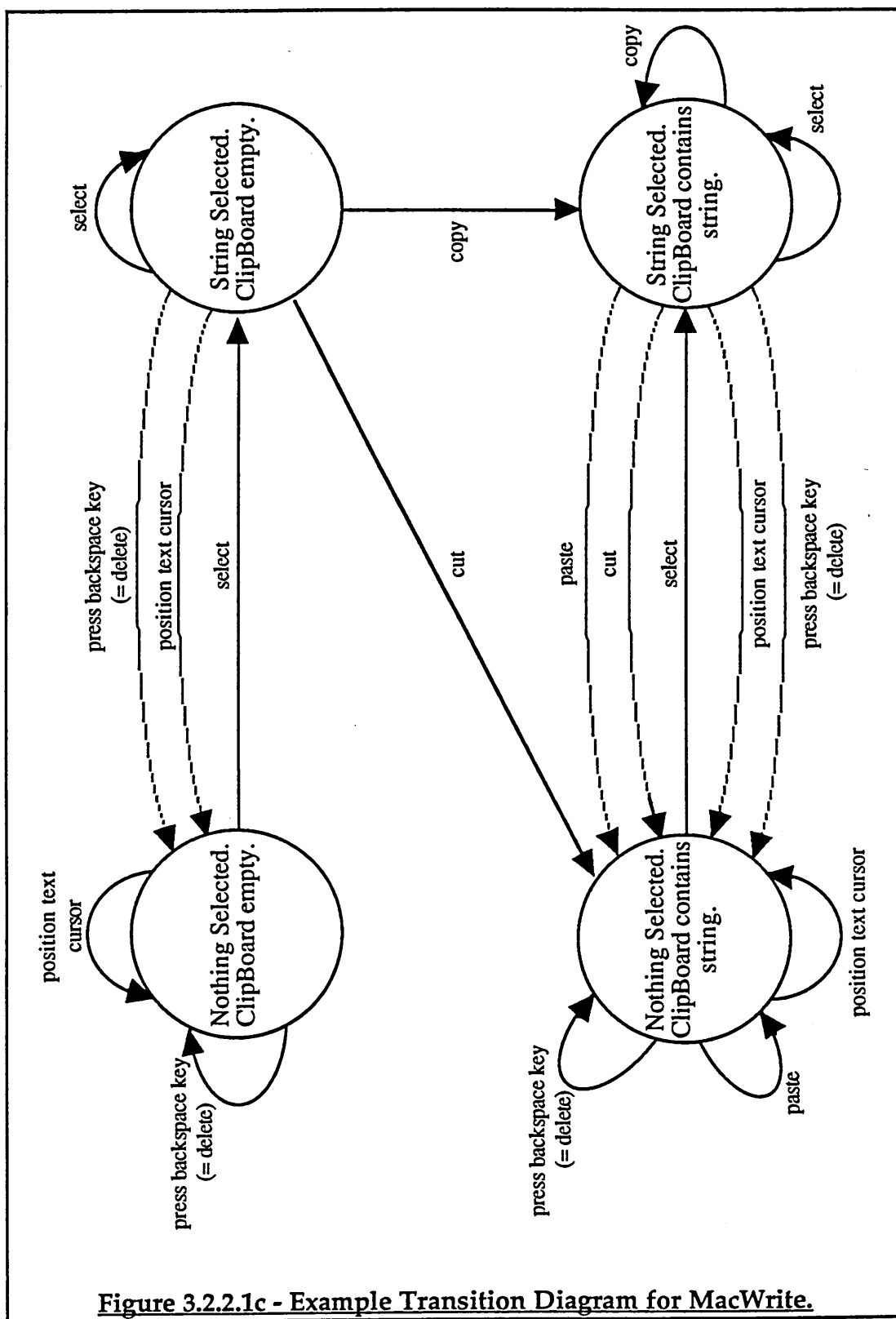


Wickens - Functional Structure of a User Interface



Norman - A General Model Of Human Information Processing

Figure 3.2.2.1b - Two General User Models.



In addition to representing procedural and structural knowledge, an analysis of cognitive strategies is useful in understanding how people control various pieces of knowledge. The most prevalent cognitive science view of human information processing is that the user is goal driven, and that task performance is directly related to specific goals. Several Conceptual Models

are based on this, especially 'Goals Operations Methods and Selections' (GOMS) [Card, S.K:1983].

Rather than provide a Conceptual Model of the user, an alternative set of design tools is concerned with developing a quantitative representation of the performance of a user at the interface. Various types of Quantitative Models are proposed which can be classified as either performance, ergonomic, or computer simulation based. Performance Models attempt to describe human performance as it relates to human information processing capabilities and limitations. For example, the Key-stroke Model developed as part of GOMS [Card, S.K:1983]. Ergonomic Models are concerned with anthropometric and biomechanical data relating to the user. Typically, these describe the physical characteristics of the user. Finally, Computer Simulation Models specify a mathematical, or logical model of Human Computer Interaction. Most Computer Simulation Models are Task-network Models, which structure the interface around the task, sub-tasks, inter-connection of sub-tasks, rules for connecting sub-tasks, and the time taken to complete sub-tasks, for example, HOS [Lane, N.E: 1981] and SAINT [Chubb, G.P:1981]. Chapter five presents an example object oriented Quantitative User Model implemented in Smalltalk 80. This model attempts to quantify system usage in order to assist error handling and advice giving.

In essence, User Models play an important role in designing interfaces which 'fit' the user. The current research issues centre upon :-

- identifying relevant user knowledge associated with the Human Computer Interaction
- developing formal model specification methods [Hoppe, H.U:1985],
- developing models which capture user differences
- developing interface design methods which capture and use this knowledge
- development of Intelligent Interfaces which can directly interpret these User Models, infer changes which occur in the user, and update the models accordingly.

In the near future, it is probable that stereotypic User Models will be used to adjust the interaction style and dialogue of interfaces. In the more distant future, individual users may have their own computerised User Model

which describes their individual characteristics and preferences. This model can then be used by different Intelligent Interfaces as they move between different systems. Each Intelligent Interface will be capable of interpreting and updating this model accordingly, as the particular application is used.

3.2.2.2. Application Model.

These models serve to describe the functionality of an application [Adhami, E:1987]. Such models are necessary for the development of separable Intelligent Interface architectures. In order for the interface to communicate with the application in a defined way, the application must describe its structure. This structure can then be linked to user interactions at a higher level, from which the user interface is built. The Application Model comprises of this description. Indeed, the Application Model must at least describe the functions, functional contexts, sub-function structures, functional side-effects, and reversibility of functions. Further requirements for Intelligent Help and Planning modules include function documentation, and goal and task strategies which link functions to tasks and eventually to user goals.

An application which was developed using a formal methodology should already have some form of explicit Application Model defined, for example Data Flow Diagrams [Weinberg, V:1979]. Applications which were not specified using these formal methods, will require some form of task analysis to generate this model [Samurcay, R:1987].

Several formal grammars are already proposed as a means to describe Application Models. Command Language Grammar (CLG) [Moran, T.P:1981] provides a framework for describing applications. This framework is essentially an ordered set of descriptions, each description being at a different level of abstraction. The same basic notation is used at each level. The four levels used by Moran cover the tasks which the user brings to the application (Task Level), the objects and procedures manipulated by these tasks (Semantic Level), the Command Language available (Syntactic Level), and the dialogue involved when using the application (Interaction Level). The levels are then connected by means of mappings across adjacent levels, e.g. the task level descriptions are linked to objects and procedures at the semantic level. This, and further formal grammars are discussed by Clowes [Clowes, I:1985], Fountain [Fountain, A.J:1985], Green [Green, T.R.G:1988], Hoppe [Hoppe, H.U:1985], and Hufit [HUFIT:Overview].

To conclude, Application Models potentially assist the design of interfaces in many ways [Totterdell, P.A:1986b]:-

- consistency checking
- efficiency checking
- integration cross referencing
- orthogonality checking
- Intelligent Help
- interface separation.

3.2.2.3. Real World Model.

Certain other knowledge, beside that which is application or user dependent, is required by an Intelligent Interface [Totterdell, P.A:1987]. This includes knowledge concerning technical definitions, conversion formulae, and the inter-relationships between similar applications especially functional, or command equivalence (e.g. DIR and LS commands provide directory lists in MSDOS and UNIX accordingly). This knowledge must also be made available in a computer usable format, and is necessary for Intelligent Help and Advice Giving modules.

3.2.2.4. Summary.

Although various types of modelling technique are now beginning to influence the improvement of interface usability, many issues still remain unresolved [Pratt, J.M:1987]. Current models tend to be too general to be applicable to specific interface designs. The level of detail in the models must be increased, and methods of analysis for defining these detailed models must be improved. Further work is also required on model validation. The accuracy and validity of models must be determined, and the limits of their representation specified. Theoretical and empirical research is needed to uncover the behavioural correlates of various knowledge representations. Similarly, comparisons are needed so that the correct models can be matched to different design processes and Intelligent Interface types.

Further empirical work is required to test the usefulness of modelling techniques as a means for improving the user interface. Before modelling

techniques become established as a tool for the interface designer, research is needed to examine their potential and limitations within real applications.

The models discussed here can be difficult to define, and even more difficult to implement in a formal computerised form. Unless models completely describe their problem domain, they cannot accurately predict knowledge or events in their real counterpart. User Models are particularly problematical [Sutcliffe, A.G:1987]. User characteristics and needs are complex, and it is often impossible to generalise from specific experiences. Individual users can also be unpredictable, constantly changing, motivated in different ways, and affected by a wide range of factors. However, work aimed at defining and implementing computerised models should not be dismissed. Such research offers insight into the diverse Human Computer Interaction influences. At the other extreme, models are not the complete solution to improving interface usability, and there are many other influences which need to be addressed.

New interface architectures are required which can maximise the potential of explicit modelling representations [Browne, D.P:1987]. Complete interface design methodologies are also needed, rather than isolated, independent, and often fragmented methods. New interface design methods must attempt to formalise the interface specification to fit these new interface architectures. They must focus attention upon the users, and their individuality. Such methods must also be integrated with existing (or possibly new) Systems Analysis and Design methods, enabling complete applications and interfaces to be developed in unison.

Modelling draws upon techniques from both Artificial Intelligence and Cognitive Science [Gilbert, G.N:1987]. Artificial Intelligence provides insight into knowledge representation - e.g. expert systems; and inferences based upon this knowledge. It also provides solutions to goal recognition and discourse modelling. Cognitive Science provides a user centred approach to modelling, and helps determine precisely what should be modelled. It provides techniques for understanding how a user learns and retains knowledge concerning computer applications and their associated domain. It also provides an insight into how a person models the real world. The closer a system parallels the way a person models the world, the more chance it has of being predictable to the user.

3.2.3. Adaptive Interfaces.

One way of dealing with the problem of multiple users and their changing requirements is to provide adaptive interfaces. Currently there are many adaptable interfaces which can be customised for individual users [Minor, S:1987], [Trigg, R.H:1987]. Adaptive systems serve to automate this customisation, which can therefore take place continuously as the application is being used [Adhami, E:1987], [Fowler, C.J.H:1987]. This adaptive function must have an overall objective [Cooper, M:1988]. Some possibilities are listed :-

- adaption to improve accuracy
- adaption to increase interaction speed
- adaption to reduce errors
- adaption to increase a users understanding of an application.

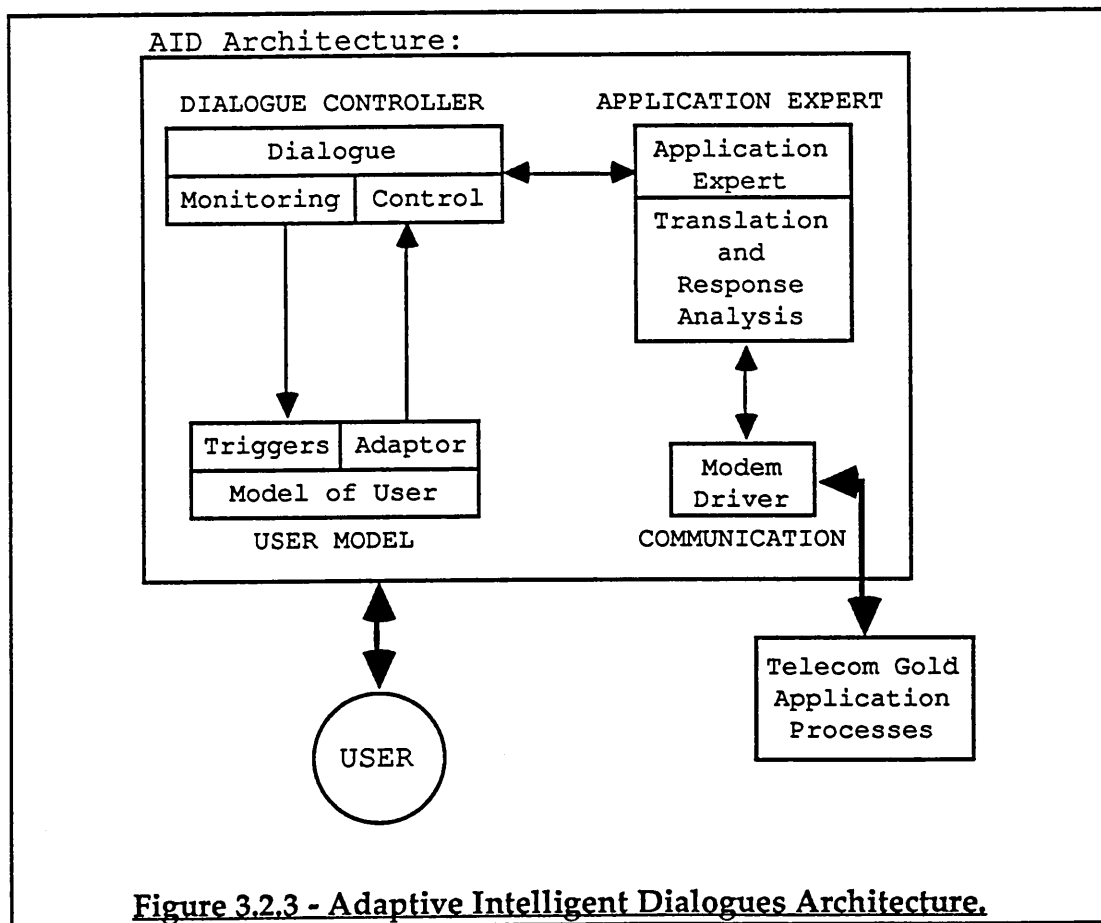
The prime objective must always be to improve the usability of an interface, and reduce the mismatch between User and Application Model. Adaptive systems must know what can be adapted, and how and when to adapt. Finally, they require a model of individual users in order to maintain adaptations for different users across interaction sessions.

Adaption should apply strictly to the interface. Adapting the functionality of an application would not only be extremely difficult, but may also violate any Systems Analysis and Design methods which were used to generate it. Modification of functionality could also endanger consistency and integrity. Application functionality must remain constant between different users, as it serves as the structure onto which an interface is built. This functionality may be incorrect, which points to a failed or misused Systems Analysis and Design process, but this failure needs to be addressed by application re-design.

It is no use adapting to features which remain constant throughout the user population. Deciding what can be adapted is therefore a question of identifying what variations exist between the interaction requirements of different users. How and when to adapt are difficult questions to address. For each specific adaptive feature, an Intelligent Interface needs to know the range of possible values, and the relationships to other adaptive features. A set of adaption heuristics must also be maintained which match the respective feature values to different users. This is a problem which may

best be tackled within the scope Artificial Intelligence and Cognitive Psychology, and assumes that user differences can be identified and defined using knowledge bases, heuristics, and logic. Several adaptive systems already exist [Croft, W.B:1984], [Greenberg, S:1985]. A major adaptive system development which has provided an insight into this difficult area is the Adaptive Intelligent Dialogues (AID) project sponsored by the Alvey Directorate in Man Machine Interaction [Hockley, A:1986].

The AID project set itself the objective of developing an adaptive front-end user interface to the British Telecom Gold electronic mailing system. The final system adapted along a number of selective adaption dimensions: level of guidance, context switching, recognition of analogous mail systems, and user tailoring. A generalised adaptive architecture based upon a dialogue controller, User Model, and application expert was formulated, and this is shown in figure 3.2.3 [Totterdell, P.A:1986].



During its lifetime, the AID project demonstrated the great difficulty involved with adaptive interfaces. It identified problems concerning which interface components to adapt, and concerning the elicitation and use of knowledge about the user, which is required in order to control adaption.

The project discovered that adaption could be used to achieve many different objectives. However, adaption to fulfil one objective often conflicted with adaption to achieve another. For example, adaption to improve accuracy conflicted with adaption to improve efficiency. The need for separable interface technology was also recognised, but not addressed. New user centred design methodologies were developed, a classification system for adaptive interfaces proposed, and an extensive insight provided into the requirements and potential of adaptive software.

The project failed to improve the 'usability' of the Telecom Gold system, and in actual fact made it more difficult to use [Durham, T:1988]. This raises an important issue relating to the application of Intelligent Interface technology. The initial Telecom Gold system was difficult to use and could have probably benefitted from the application of Non-Intelligent Interface technology. This includes improving interface consistency, support of multiple interface styles, better command names, and on-line help. Instead, Intelligent Interface concepts were applied without due consideration to Non-Intelligent aspects. The result was an Intelligent Interface which was more difficult to use than the original Non-Intelligent counterpart. This experience illustrates the fact that Intelligent Interface technology must be used carefully.

Beside implementation difficulties, many ethical and psychological problems also arise from the use of adaptive interfaces. With legislation on database protection now in force, User Models will need to be registered under database protection laws. Therefore, users must be allowed to access these models in order to correct any misrepresentations. This requires that the User Model is explicitly defined, separated, and is itself customisable. Effectively, adaptive systems may become customisable systems, where customisation is automated and confirmed by the user. Each time an adaptive system decides to transform itself, it may therefore have to confer with the user. This places an interaction overhead upon the user, novice and expert alike, and assumes that they understand the adaption and confirmation process. If adaption confirmation is not used, then it is possible that an interface may adapt incorrectly and cause an incorrect internal User Model.

Traditionally, people adapt and machines either remain constant or can be manually adjusted. That is, control remains in the hands of the user. If a user tries to adapt to an interface which is itself trying to adapt to the user, a

complex recursive situation may occur, and the user's model of the system will become unstable. Assuming that the interface and user can agree upon who, or in the case of the interface, what, is going to adapt, several other issues arise due to human nature [Bullinger, H.J:1987], [Nebeker, D.M:1987] :-

- the possibility of monitoring and reporting upon user efficiency may cause mistrust
- the question of correct adaption and its assurance
- adaption reversibility when things go wrong.

Unless the User Model provides a perfectly complete and accurate representation of the user, adaptive systems can never function correctly. An apparent model mismatch for one user may be a short-cut for another. Therefore, adaption is in itself subject to individual user characteristics, and ultimately to higher levels of adaption. In certain situations involving expert users, it may also be possible for a conflict to occur whereby the user tries to 'out manoeuvre' an adaptive interface.

Although many problems need to be addressed, simple restricted adaptive interfaces may be of great benefit to the user, especially where adaptation is applied to the field of advice giving and error handling [Carroll, J.M:1987]. Adaptive interface research also provides an insight into Human Computer Interaction, which is useful for non-adaptive user interfaces.

3.2.4. Planning Aids.

Human planning is essentially an activity concerned with the ability to identify an objective, or goal, and construct a sequence of actions with which to accomplish it. The sequence of actions is known as a plan, and may itself contain smaller objectives which have their own associated plan. Humans typically already know many plans for accomplishing various goals, and therefore planning is both a 'top down' and 'bottom up' process, with existing plans being re-used and new ones learnt.

When using a computer system, users also have specific tasks that they wish to accomplish, for example adding a new customer to a database, or printing out the salary cheques. To satisfy the task in hand, a user must execute a sequence of application functions. In doing so, users construct a plan based on their knowledge of the application, and previous experience with other computer systems. The plan is then executed through the user interface, and

the relevant task is completed. This planning process is most prevalent when using computer applications which are themselves designed to support human planning activities, for example Project Planning and Management Support Systems [Wiest, J.D:1977], and Military Tactical Planning Systems [Noah, W.W:1986].

Computer systems have many levels of abstraction and different degrees of functionality. To accomplish a certain task users must 'navigate' their way through a system to a certain level, and then perform a particular sequence of operations in a pre-defined order [Andriole, S.J:1986]. Two issues arise from the effect of planning upon the use of computer systems. Firstly, the ability of a user to remember where they are in a sequence of actions, what they have already done, and what is left to complete. Secondly, the ability of a user to look forward to an objective, and plan the subsequent actions required to arrive at that goal. Included in this second concern is also the ability to remember past plans, which can then be applied repetitively.

It follows that an awareness and support of human planning activities should improve an Intelligent Interface, making it easier to use [Hecking, M:1987]. Simple facilities to show the user where they are in a particular system, and how that position was achieved are essential, especially where large applications are concerned. Intelligent Interfaces should also enable users to return to a previous position, and undo any intermediate effects. These extra facilities may make use of graphics, and require that an Application Model be available.

Intelligent Interfaces may also provide more complex planning facilities [Hagendorf, H:1987]. An interface should be able to remember, or record, particular sequences of actions which constitute a specific plan, and 'replay' them to users at their request. A library of known user plans can then be maintained. Interfaces should also be capable of recognising inefficiencies in a particular plan, and able to suggest an alternative improved sequence of actions.

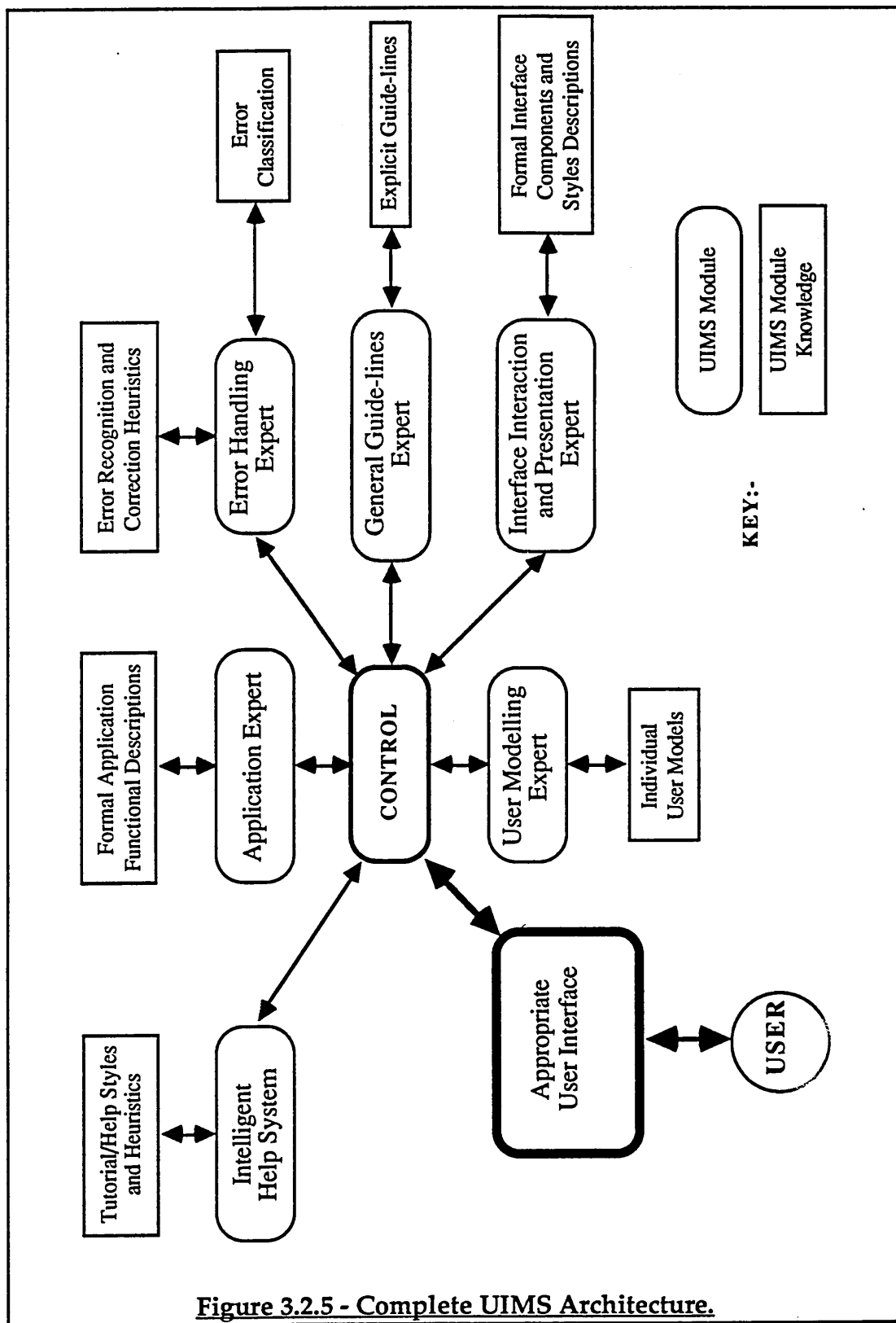
Modelling of user plans and goals should also be of benefit to Intelligent Help, and User Modelling modules of an Intelligent Interface [Carberry, S:1988]. If an interface can predict the goals of a user from the current interaction, then it may be possible to specifically tailor the available help facilities. It may also be possible to adapt the interface automatically for

a user, and complete the task in hand by automatically performing the remaining plan actions.

Plan and goal recognition is closely related to Intelligent Help, User Modelling and adaptive interface research [Desmarais, M.C:1987]. It also draws from expertise and research from within the Artificial Intelligence and Behavioural Psychology disciplines. Current research is mainly concerned with inferencing mechanisms which can elicit knowledge concerning the goals and planning processes of a user, based upon their current and previous user interactions [Pollack, M:1986]. Research is also being undertaken to develop new methods of representing this knowledge within the user interface.

3.2.5. General Architecture for an Intelligent Interface.

Figure 3.2.5 illustrates a general architecture for an Intelligent Interface, which draws together the different applications of Artificial Intelligence discussed above. It shows the various modules of an Intelligent Interface, and their knowledge requirements. Different Intelligent Interfaces may implement these modules in various ways; this diagram does not attempt to suggest the best implementation.



The various modules and their functions are summarised :-

Intelligent Help System

This handles automatic, context sensitive, user tailored on-line help and tutorial support for different applications. It requires

knowledge on tutorial and help styles, and application structure.

Error Handling Expert

This handles all errors. It requires knowledge concerning error recognition, error classification, error correction, application structure, and documentation.

Application Expert

This module describes the application structure and handles enquiries from other modules. It requires knowledge concerning the formal structure of an application.

User Modelling Expert

This deals with personalized knowledge for individual users. It maintains knowledge concerning their individual preferences and characteristics.

General Guide-lines Expert

This module maintains knowledge concerning general guide-lines such as keyboard repeat rates, and minimum response speed. It allows such knowledge to be modified and queried.

Interface Interaction and Presentation Expert

This formally describes the interface components, and how they can be combined to create the final interface, for example, windows, buttons, switches, and valid interactions. The same components can then be combined in different ways to generate other interfaces. The formal interface description can then be executed in order to generate the actual application interface.

Control

This lies at the centre of the architecture, and brings the different modules together. It provides communication between modules, and presents the final interface to the user.

In order to enable the knowledge contained in individual modules to be manipulated and viewed in various ways, it is necessary to develop specialised interface design tools. The task of interface design is now centred

upon determining the knowledge required by each module, and the Tool-set can then be used to specify this knowledge. As this knowledge is modified, so new or variant interfaces are generated.

Fundamental to this architecture is the need for interface separation and formal module descriptions. This research is primarily concerned with software requirements for separation, and how these requirements can be met with new software architectures.

3.3. Interface Classification.

From the preceding investigation it is possible to propose a broad classification of interfaces according to their adaptability and design method. Figure 3.3 illustrates the relationship between different interface design methods and the types of interface which they may be used to generate. The terminology is defined :-

Fixed.

A Fixed interface is one which is designed with a particular user, or group of users in mind. Once implemented, its structure and features remains constant (unless future versions are released). This class of interface is presently the most common.

Adaptable.

Adaptable interfaces can be configured to a particular user, or user sub-group [Trigg, R.H:1987]. This configuration can take place at any time during the use of the interface. The configuration must be performed by the user, or a trained person who has an understanding of the individual characteristics and preferences of the user. The granularity, and scope of this configuration may vary, and some interfaces may need re-compiling after modifications are made.

Adaptive.

This type of interface behaves like an intelligent observer which automatically adapts the interface to the habits and expertise of a user, without being too obtrusive [Totterdell, P.A:1987]. Again the granularity and scope of this automatic

adaptation varies, and ultimately interfaces will allow the adaptive mechanisms to adapt.

Generic Interface Design.

As opposed to Application Specific Interface Design, this approach re-uses existing interface components and interaction dialogues within interfaces for different applications [Kraak, J:1987]. The actual interface structure may differ between applications, but the same interface features and components are re-used. This term is best applied to the design Tool-set, rather than the final interface that results.

Interface Framework design.

This type of generic interface design (and its associated Tool-set) packages the code that implements most of the user interface into a reusable, and extensible skeleton [Coutaz, J:1987]. The designer's task consists in filling the blanks of the skeletons, adding new functions, or overriding parts that do not fit the application domain.

User Interface Management Systems Design.

With this generic approach, the interface designer describes the interface in a pre-defined language. This description is then used by a User Interface Management Systems (UIMS) to automatically generate an executable interface for the user [Alty, J.L:1987]. Modifying and prototyping new interfaces is simply a case of changing the description and re-compiling the new interface.

		Design Method / Tool Set		
		Application Specific	Generic	
			Framework	UIMS
Interface Type	Fixed	✓	✓	✓
	Adaptable	✗	✓	✓
	Adaptive	✗	✗	✓

Figure 3.3 - Classification of Interfaces.

The UIMS design approach is potentially more powerful than the Interface Framework approach, which in turn is more powerful than the Application Specific design approach. However, the power of these approaches lies in how well they are used. A well designed Application Specific interface may out perform a poorly designed UIMS interface, in terms of 'usability'. Similarly, a well designed Fixed interface may be more usable than a poorly designed Adaptive one.

As figure 3.3 illustrates, certain types of interface require certain design approaches. Each approach has an associated Tool-set for use by the interface designer. In the case of Application Specific interface design, this Tool-set is normally part of, or an extension to, the application implementation language. With the case of Generic interface design (UIMS, and Frameworks), this Tool-set is usually separated to some extent from the application, resulting in a more distinct interface. Examples of Frameworks include the Graphics Environment Manager system, where extensions to the implementation language are provided. UIMS provide the most distinct form of application and interface separation, where the interface design Tool-set is normally completely separate from the application language,

3.4. Approaches to Interface Design.

From the viewpoint of the designer, inseparable Application Specific user interface design is probably the simplest approach. Although some degree of separation may exist, this approach does not distinguish between application functions and the user interface. The application functions define their own

interface requirements. These are implemented using the application implementation language, or an extension of the language. Little attention is usually given to the user interface, which is simply mapped onto the input / output requirements of the application functions. This approach may be suitable for simple bespoke applications, but is insufficient for more complex ones.

An alternative is the use of Generic design methods. With this approach the interface is separated to some extent from the application functions and implemented using special languages or tools which can be distinguished from the language and tools used to implement the application function set. One Generic approach is the framework approach, where a language or system provides the framework for the final interface. The designer's task is to select the necessary interface components, and fit them into the framework provided. The designer is constrained to this framework, and must design accordingly. Because of the constraints imposed by frameworks, the application functions may often have to be designed to fit the interface requirements. This will have the effect of binding application functions to specific interface components, thus making interface and application re-design more difficult.

A UIMS is potentially the most powerful generic design method. It is similar to the framework approach, but the interface constraints are more relaxed. This is because a UIMS specification language is more flexible and expressive than a framework. The granularity and restriction imposed by a particular UIMS specification language ultimately determines the potential of a UIMS. This specification language effectively maps low level user interactions, such as key strokes and mouse movements, onto high level concepts such as goal and task modelling. As the UIMS language becomes more constrained, so a UIMS moves towards a framework approach. This results in a grey area between UIMS and framework approaches, which is difficult to classify.

Applications are based upon information and tasks. Tasks can be applied to information in order to enquire upon, modify existing, or create new information. Similarly, certain types of information can only be acted upon, or used by certain tasks. A computer application models some physical or conceptual real world system, or sub-system. Meanwhile the interface presents these applications to the user. To use a software engineering term, the interface is adhered to the application function set. If this adhesion is too strong, then only certain interface components can be attached to certain

types of application functions. Thus the interface has a strong effect on application functionality, which must be designed accordingly.

If the adhesion is too weak, then any type of interface can be built for any application. The resulting interface may misrepresent the application functionality, and therefore present an incorrect Application Model to the user. For example using a Bar Chart to display textual rather than numeric values. Assuming that this is possible, designers would have to constrain themselves so that the final interface correctly represents the application functions to the user.

The correct adhesion, or separation, lies somewhere between these two extremes. A preferred intermediary is where the application constrains interface design so that it cannot be misrepresented to the user. At the same time it must allow a certain amount of freedom for different interface components to be used to represent the same application functionality.

3.4.1. Requirements For Good Interface Design.

There are two primary goals to be met by interface software design methods and related Tool-set :-

- ease of use of the final user interface
- ease of use by the interface designer.

Fortunately, these goals reinforce one another and parallels can be drawn between the two. Although easy to use design methods cannot guarantee an easy to use interface, it stands to reason that they can improve and clarify the design process. Subsequent user interfaces should then reflect the quality of such design methods. The major objectives, problems, and solutions concerned with improving the usability of computer software identified so far can be summarised :-

Separation

An essential feature which ultimately determines the potential of Intelligent Interfaces.

Formal Descriptions

Required by different Intelligent Interface modules, and in different formats.

Expert System Modules

The division of an Intelligent Interface into smaller communicating Expert System modules. This provides focal points for research, and aids Intelligent Interface maintainability.

These criteria provide many advantages. Separation frees the interface designer to concentrate on the interface alone. Various types of interface may also be easily prototyped, and experimentation encouraged.

Formal descriptions should provide a formal 'backbone' to interface design. New interfaces are effectively generated by altering existing, or creating new formal descriptions. The interface designer can observe the effects of these changes, learn by experience, and repeatedly apply the same changes within other formal interface descriptions. The use of formal descriptions should also enhance interface consistency and integration [Bez, H.E:1987].

The use of distinct Intelligent Interface modules should assist the interface designer by providing a framework within which to build the interface. The interface designer can then focus attention on various interface modules, and observe the effects of any modifications. Ultimately, interface designers may specialise in different interface module areas.

Further requirements for the interface designer include :-

- code re-use
- immediate feedback from changes to formal descriptions
- support of different interface design levels
- interactive Tool-set (i.e. Designing Systems by Example [Dearnley, P.A:1983]).

Code re-use is a well understood software engineering term [Bell, D:1987] and can be applied to Intelligent Interface design approaches and support tools. For Intelligent Interface design, code re-use necessitates formal interface descriptions within different modules. Particular interface styles, features, and knowledge, could then be easily re-used within other

interfaces. A library of generic interface components can then be maintained. These components may employ default values which can be customised for individual interfaces.

The effects of changes to the formal description of an interface should be seen immediately. As response time affects user acceptance of an interface [Thimbleby, H:1986], so the Tool-set response time affects the acceptance of the Tool-set by an interface designer. A fast Tool-set response time should encourage experimentation with different interface styles. This should enable the interface designer to be more creative, and hopefully design better interfaces.

The design method and Tool-set should support different conceptual design levels [Kraak, J:1987]. For example, a key-stroke level which maps user interactions onto individual interface component tasks, and a presentation level which maps the interface component output onto a virtual graphics window. An interface designer can then construct a complete interface from the various interface components. In doing so, the designer does not need to consider the key-stroke or presentation levels. This is coupled with component re-use, whereby individual interface components can be selected from a library with default key-stroke and presentation levels. Effectively, an interface designer may interact with the Tool-set at different levels according to their expertise and objectives.

The interface design Tool-set should provide an interactive Tool-set which automatically maps onto the underlying formal interface description. The interface designer may then implement an interface using either an interactive Tool-set, or by directly specifying the formal description. The Tool-set should support a 'Design by Example' approach. The interface designer would implement the final interface by interactively placing the interface components on the screen, as they are to appear in the final interface. Interactive tools should also be provided for modifying lower component levels, and for other Intelligent Interface modules. The Tool-set is in effect an extension of the UIMS, which produces the final Intelligent Interface from the formal description. It serves two purposes; allowing interfaces to be interactively implemented, and also generating the appropriate interface formal description upon request. An interactive Tool-set may also be self describing. Its own interface may itself be generated from a similar UIMS Tool-set. Effectively, the same usability features which will

eventually be embodied into the final application user interface can be incorporated into itself.

3.4.2. Graphics Environment Manager.

The Graphics Environment Manager (GEM) system was developed by Digital Research for the IBM PC range of microcomputers [Bright, P:1988]. Its purpose was to provide a user friendly operating system and graphics software routine library. The operating system offers a window and mouse based iconic graphic interface as an alternative to the traditional command driven system. The graphics library provides a set of machine independent routines to support standardised 'user friendly' interface features such as windows, mouse pointer movement, icons, and drop down menus. These routines can then be used within application programs, and are accessed using function or procedure calls. Various implementation languages are supported, and suitable binding files are available for purchase. The complete GEM system is easily re-configured to support different hardware devices, and this does not affect the execution of application software using particular library routines. Several Desk Top publishing applications are also supplied as part of the GEM system. Figure 3.4.2 illustrates the basic GEM software architecture, and shows how an application uses individual library routines.

The GEM system is a useful Tool-set for the interface designer, and provides a set of standard re-usable interface components from which to build a complete interface. GEM distinguishes between the interface and application. However, this separation is based upon two components; the interface and the application. Dialogue control is maintained within an application program, which replaces its usual input / output statements with calls to the interface routine library. Effectively, the interface components are 'strung' together with the thread of dialogue control remaining within the application. In order to modify an interface, the application program code must be modified and re-compiled. This restricts the flexibility of the approach, as interface knowledge is contained within the application. Similarly, it is difficult to implement multiple interfaces for the same application without duplicating the application, and varying the interface library routine calls in each copy.

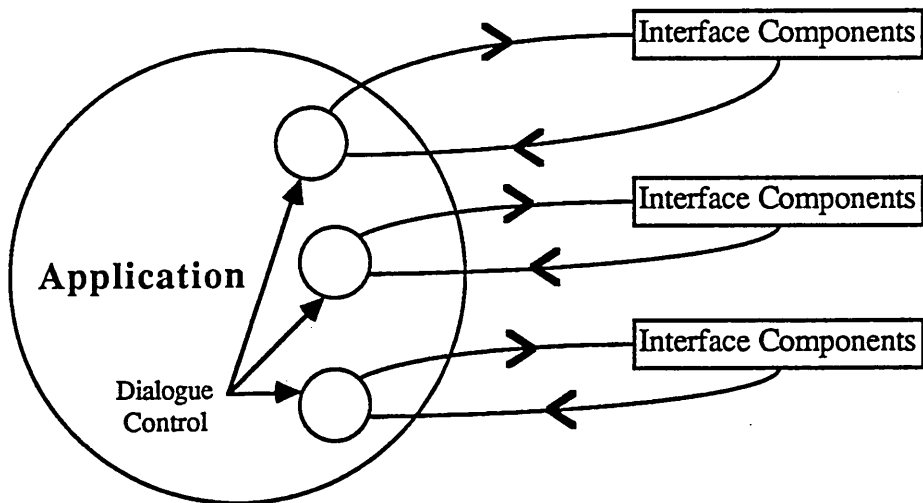
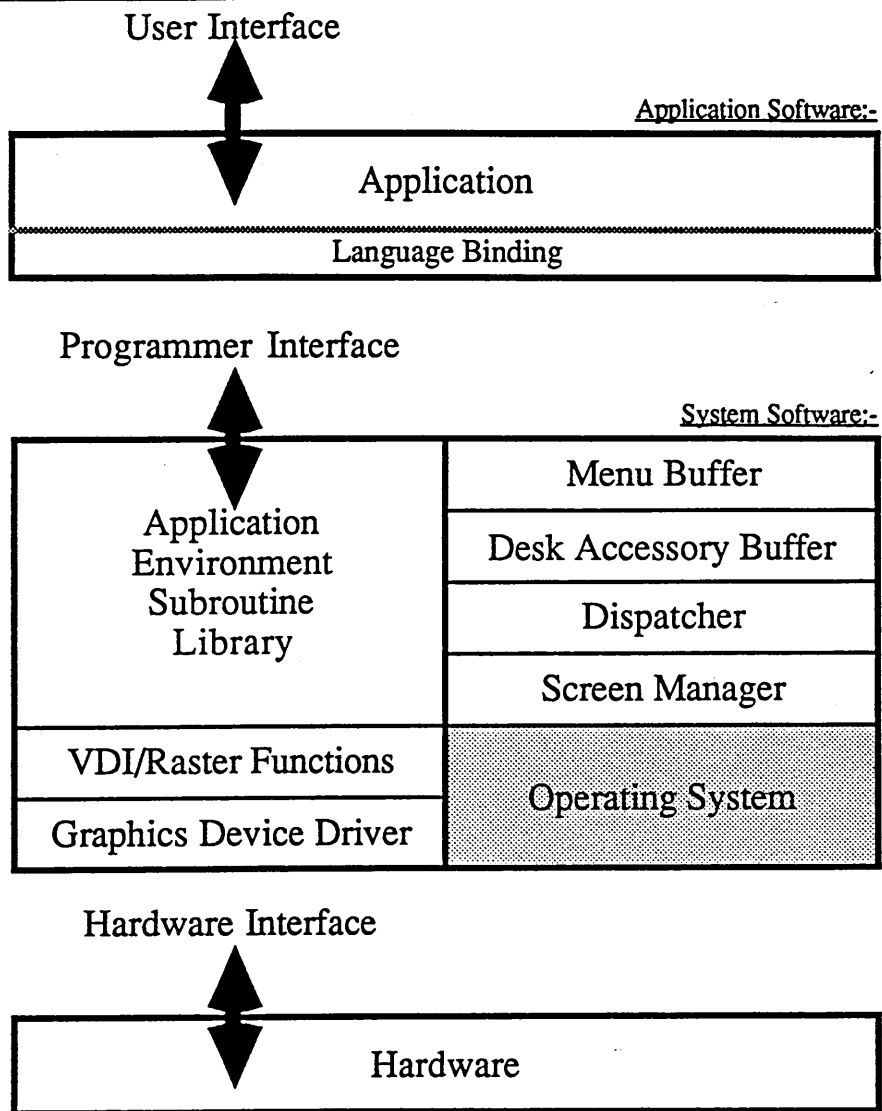


Figure 3.4.2 - GEM Architecture.

A third separation component is needed if this problem is to be overcome. As discussed later, this extra component contains knowledge which links the interface and application. This knowledge can then be easily changed in order to modify an existing interface. It also allows separate interface components to be re-used within many interfaces.

3.4.3. The Model View Controller Mechanism used in Smalltalk 80.

Smalltalk provides a built-in interface concept, namely the Model View Controller mechanism (MVC) [Smalltalk80:ReferenceGuide]. This is illustrated in figure 3.4.3, and the three components are now described.

Model.

This is the application itself described in Smalltalk code using Classes, inheritance, polymorphism, and other object oriented techniques. This is defined and tested first.

View.

This is the output interface which is seen by the user. Examples of views are all graphical windows which are displayed on the screen. These windows may contain graphics such as text, boxes, and circles. Smalltalk provides many existing Classes which cover all of the basic graphical functions, such as drawing various shapes, rotating, translating and scaling graphical pictures, and much more. These features facilitate the description of complex views.

Controller.

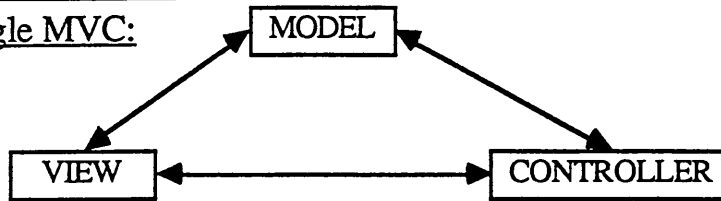
This is the input interface. It effectively maps the input functions onto application functions. Smalltalk input functions are received either from the keyboard (i.e. pressing various keys) or from the mouse (i.e. moving the mouse and associated screen pointer, and pressing the various mouse buttons).

Specific Smalltalk Classes are implemented to support the MVC mechanism. Several different MVC mechanisms can be defined for the same application, giving the user a choice of interface styles. Theoretically, new interfaces can also take advantage of existing MVC component implementations. This is achieved by code re-use and inheritance, which is

supported within most object oriented languages [Horn, C:1987]. In practice, this is difficult to achieve due to the two component separation model on which the mechanism is based. The mechanism distinguishes between interface and application functions, and dialogue control is correctly maintained within the user interface (i.e. combined View and Interaction Controller). However, the interface contains knowledge concerning application functions. This takes the form of embedded Model, or application function calls. If an existing MVC mechanism is to be re-used for a different interface, these embedded calls must be changed. This often requires considerable modification to the MVC Classes, and must be done by an experienced Smalltalk programmer. Again, the need for a third separation component can be identified which contains information which links the application and the user interface.

Problems also exist with defining the boundaries of the separate MVC components. That is, what functions should be included in the different components. For example, it is easy to include dialogue control as part of the View, rather than the Interaction Controller. Similarly, the View and Interaction Controller may easily contain application functions. The choice is ultimately left to the programmer, and may differ between various MVC implementations. Finally, no MVC Tool-set exists apart from the standard Smalltalk Class Browser. As a result, considerable programming expertise is required to implement interfaces using the MVC concept.

Single MVC:



Multiple MVC:

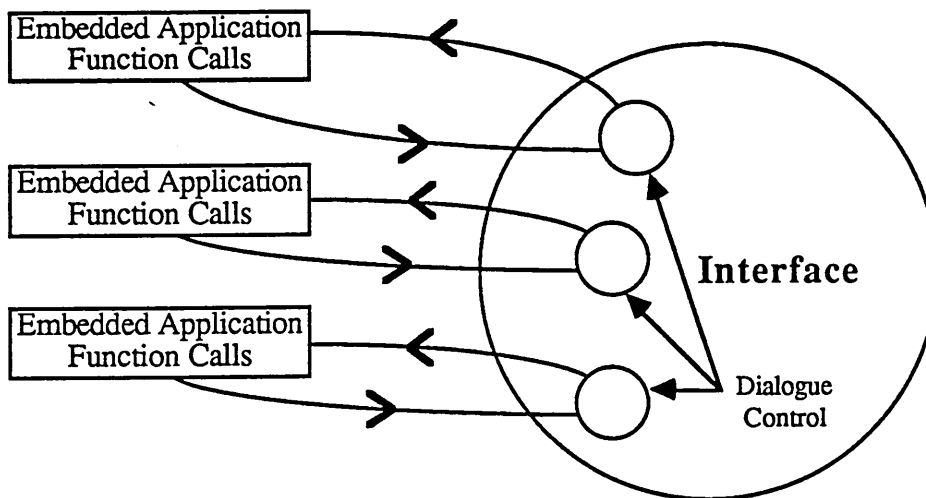
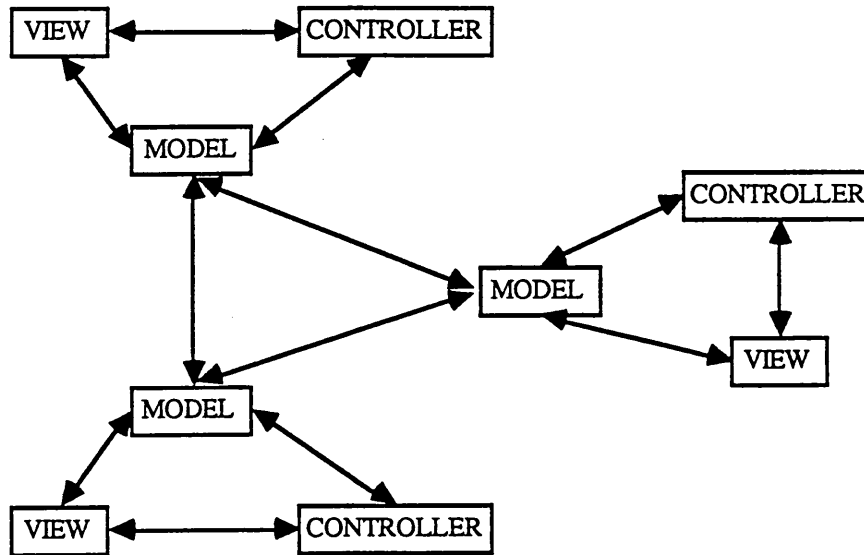


Figure 3.4.3 MVC Concept.

3.5. Summary.

Chapter two discussed the principle software influences which affect the user interface. This chapter has examined how Artificial Intelligence can be used to utilise these influences in favour of the user. The major areas of Artificial Intelligence application were identified as Intelligent Help and

Tutoring, Modelling, including User and Application Modelling, Intelligent Planning Aids, and Adaptive Interfaces. An Intelligent Interface was proposed based upon these areas, and this was summarised in figure 3.2.5.

This chapter has also examined the different approaches to interface design, and their associated Tool-sets. The requirements which must be met by an interface design approach were listed, and discussed in relation to several existing design approaches. In order for the user interface to be improved there must be a move towards Intelligent Interfaces, generated using integrated UIMS. These UIMS require distinct interface separation and should enable formal interface descriptions to be both generated and executed. Finally, they must also provide specialist tools which support interface designers in their task of designing consistent, personalized, and usable user interfaces to a wide range of software applications.

The application of Artificial Intelligence to interface design must be viewed in relation to the total impact which software factors have on the user interface. First and foremost, attention must be given to the Non-Intelligent factors listed in chapter two. The Adaptive Intelligent Dialogues project demonstrated that unless these factors are correctly addressed, an Intelligent Interface will probably make a poorly designed interface worse.

The effect of software design and Artificial Intelligence on the interface must also be considered in context of other factors which affect user acceptance of complete computer systems. These factors include Systems Analysis and Design methods, political and organisational effects, and social influences. The next chapter discusses these other effects in more detail.

Chapter Four.

Experience With Other Influences which Affect User Acceptance of Computer Systems.

4.1. Introduction.

User acceptance of computer systems depends upon many factors. These may be political, ethical, social and organisational as well as the characteristics of the interface itself. Having so far examined the various software factors which affect user acceptance, it is necessary to consider other influences.

As part of this research, an investigation was undertaken to determine the software factors which affect the 'user friendliness', and user acceptance of real computer systems. An exemplary working library database computer system was therefore selected. This investigation was intended to provide an objective understanding of the problems associated with using computer software systems, and a practical insight into the potential improvement of computer software. The investigation revealed that many other factors also affect user acceptance, and that it is often difficult to isolate specific software effects.

Knowledge gained during the early stages of the library investigation demonstrated the need for new software architectures and interface design tools. As a result, effort was directed towards the development of a suitable software architecture and Tool-set to support the design of computer systems, which reflect the actual interaction requirements of individual users. The results of this investigation are presented in section 4.2.

The experience gained during the library system investigation also provided a valuable insight into the wide range of factors affecting user acceptance of computer systems. In particular, the influence of Systems Analysis and Design became apparent. General issues and observations arising from the empirical library system investigation are presented in section 4.3. This section also discusses the possible effects of separable User Interface Management Systems (UIMS) upon traditional Systems Analysis and Design.

4.2. The Working Library System.

A large library database system and its user group were selected as the subject of a detailed investigation into user acceptance of computer systems. The main selection criteria were :-

- large user group
- recent system design and implementation
- easy access to system and user group
- full support from management and union.

During the initial planning stages links were established with Sheffield University Applied Psychology Unit (SAPU). This collaboration provided expertise within the fields of experimental control and evaluation, and cognitive psychology.

4.2.1. Overview.

The main role of the library system investigation was to provide an insight into the software factors which affect user acceptance of computer systems, and in particular the user interface. These factors could then be addressed by improved software technology. The library system user group was also to serve a secondary role as a specialised computer user group on which to test new interface software. The selected system had a user group of over 35 users from varied backgrounds, with different levels of expertise in both the application (i.e. library databases) and computer domains.

With assistance from Sheffield University Applied Psychology Unit, the following plan was constructed :-

- (1) Questionnaires to entire user population.
- (2) Notebooks left with experimental group to note everyday problems, and ideas.
- (3) Initial interviews with experimental group to determine user profile including information concerning their background, job status and content, and general attitudes.
- (4) Further detailed interviews concerning their use of, and problems with the computer system.
- (5) Video recording and analysis of the experimental group performing set tasks with the computer system.

- (6) Follow up to stage (5) with further interviews.

To assist the control of the investigation, the user group was divided into 7 experimental groups each containing 5 subjects. All interviews were to be recorded with audio tape, and any sessions using the computer were to be recorded with video cameras. It was intended that the resulting data be carefully analysed in conjunction with Sheffield University, using well established experimental analysis techniques. Stages (2) - (6) were then to be repeated again with further experimental groups. This was to help eliminate individual bias which may be present within a user group.

In fact, the investigation only reached stage (4) with the first experimental group, when it became apparent from the data gathered that the investigation of new software architectures and interface design tools was vital, in order to eliminate most of the problems encountered.

4.2.2. Library System Description.

The library computer system consisted of a mini-computer, with 20 terminals attached. These terminals supported text, and had no graphic capabilities. Several terminals also had a printer attached, which could handle screen dumps when required. The database software supported multiple users, and was specifically tailored to the libraries information requirements.

4.2.2.1. Database structure.

The database was hierarchical in structure, composed of a large, single file, with approximately 40 fields. Some fields were grouped together into repeating groups according to their function. Duplicate records were allowed in this file, but the application software prevented identical records and repeating groups from being entered.

The range of values of many of the database fields were restricted to pre-defined sets. For example, Site Name can only be 1 of 5 values, i.e. Site 1, Site 2 ... Site 5. However, this information was not available through the user interface, and users had to learn them. As a result, many incorrect or misspelt entries occurred.

The main database tool was the Pointer File. This was a file which users could create themselves, and contained a list of pointers to records within the main database. Pointer Files could be created by various methods, e.g. the results of a book search by title, or a selection on a certain author name. Once created, each Pointer File was given a name and a creation date. Many Pointer Files could exist at any one time, and were always owned by individual users. These Files enabled users to manipulate sub-sets of the main library data file. For example, Pointer Files could represent a group of books by one author, or books containing the word 'byte' in their title, or books by certain publishers.

Once created, records could not be directly added to Pointer Files, although it was possible to merge two existing Pointer Files to create a new one. Records could be deleted from Pointer Files, which had the effect of deleting the actual record from the main file. Finally, when the Pointer File itself was deleted, none of the records in the Pointer File were physically deleted from the main file.

4.2.2.2. User Interface.

The user interface was basic, and was primarily command driven. After logging on with their user codes, users were presented with a single menu containing six items, and prompted to type in a selection. The system responded with a single one line message describing what part of the system the user was in, and prompted them to enter a command. Once a correct command, or abbreviation was typed, the user was prompted for further information, using a single line question and answer interaction style. Incorrect commands resulted in a basic 'no such command' error response. When necessary, an example database record format was shown on the screen. This comprised of field titles, and field values. Pressing the cursor keys moved the cursor between field values, and pressing the return key accepted the screen in its current state. By typing in different information in the appropriate fields, specific records were selected for use by the current task. After the last prompt the system initiated the command, and either returned to the command line prompt, or displayed the required database information. Users could quit to the previous level, i.e. command or main menu at any time, using the quit command. They could also leave the system by making the relevant main menu selection.

Help was basic and only available at the command line prompt. It consisted of lists of possible commands, or lists of possible field names.

4.2.3. Library Investigation Results.

This sub-section presents the results collected during the completion of stages (2) - (4) of the investigation, for the first experimental group. It is recognised that these experimental results came from a small subject group, and may not be typical of all computer users, however they should not be ignored.

4.2.3.1. Initial Notebook Investigation.

According to the plan detailed in section 4.2.1, stage two of the investigation entailed the distribution of blank booklets to each subject. Subjects were asked to describe points of interest regarding the library database system, and any other comments they felt were relevant.

The response to this was varied. Three subjects were helpful, describing in great detail many problems with the system. The remaining two expressed problems because of lack of time, and a preference for verbal descriptions. Appendix A contains a list of statements collected during this initial investigation by the three participating subjects.

The main user interaction problem areas identified by the participating subjects were as follows :-

- unforgiving environment
- inconsistencies between field names, and meaning
- insufficient, and non-context sensitive help
- slow response time
- insufficient, and unforgiving help messages
- no apparent logical ordering of records
- inconsistencies between command syntax within similar tasks
- unable to switch context and perform another task while maintaining current state of system
- lack of continuity of tasks. Previous tasks cannot always have an effect which is desirable for a later task to use
- cognitive problems understanding the applicability of actual information in the computer

- problems transferring knowledge between different systems.

Although there were other problems mentioned relating to the areas of organisational structure and office environment, these are only acknowledged as they were not studied in any depth.

4.2.3.2. Initial Interviews.

Stage three of this investigation involved a detailed background interview with each participant. This was aimed at gathering background information about each person regarding previous experience with computers and library information systems, job description and daily tasks, amount of work done using computer systems, attitude towards computers, and initial comments on the library system. The main objective was the specification of individual user profiles. These could then be used to help understand their various comments throughout the remaining investigation.

Because of the change in direction made after the completion of these interviews, the original recorded transcripts are not included. However, several interesting issues arose.

Experienced computer users appeared to have fewer problems with the system. They were more able to deal with inconsistencies, and were apparently used to interacting with difficult to use computer systems. Inexperienced users were less tenacious, and quicker to blame themselves for any difficulties rather than the computer system. Unfortunately this created a difficult situation, whereby inexperienced users found it difficult to progress because of their inexperience.

Stereotyping was a problem; One member of the group was very experienced and was branded a 'computer boffin'. Other group members did not want to 'end up like him' and only used the computer system when it was essential. Several users also felt threatened by the introduction of new computer technology.

Users found it difficult to distinguish between problems caused by poor interface and application functional design. They simply saw a complete computer system and could not classify the difficulties encountered with the system.

One subject who disliked the system and found it difficult to use, expressed the opinion that they knew how the system was chosen, and implemented. This selection and implementation process was, in their eyes, inferior, and the computer system was forced upon them. In their opinion the resulting system was a second rate product of this inferior design process, and therefore was not acceptable.

Most subjects saw the potential capabilities of computers in the future, and some looked forward with great enthusiasm to new applications of computer technology. Others were more cynical, and saw computers as a 'necessary evil'.

4.2.3.3. Further Interviews.

After further interviews with members of the library subject group, the following issues were discovered and discussed.

Most users of the library system (not only the subject group) had conceptual problems with the manipulation of Pointer Files. Users were prevented from directly deleting records from the main file, and understood the reasons for such protection. Pointer File deletion was, however, allowed. Also, when the Pointer File itself was deleted, none of the records in the Pointer File were physically deleted from the main file. This was conceptually different from deleting individual Pointer File records. Similarly, they could not understand why additions were allowed with the main file but not with Pointer Files. The Pointer File appeared as a sub-set of the main file, yet the same tasks could not be done against it. These problems with Pointer Files were due to inconsistencies within the application, rather than the interface.

On-line help for the system was minimal, and only available at the command level. A help option was available from the main menu, but gave the response that help was not yet available. This suggested to some users that it never would be. Expert users found the help useful, but novices found it insufficient.

The documentation for the system was difficult to read or understand, and was obviously aimed at technical experts. As a result, user training was on a person to person basis and most problems were sorted out by asking a more experienced user.

Abbreviations in the system were inconsistent at different levels, with one letter meaning one command at one level and a different command in another part of the system. Also, in certain parts of the system abbreviations were not allowed. This problem was due to inconsistencies within the interface itself.

The system allowed users to make only three consecutive errors when answering the prompts for data that a command required. It then returned the user back to the command prompt. Users often used negative responses to interrogate the system. For example, searching for non-existent books resulting in a 'record not found' message. After several consecutive 'not found' searches, which the system treated as actual errors, the user was returned to the command prompt. As a result, they had to traverse back to the same database search screen before continuing with further queries.

There were still some obvious bugs in the system. These bugs were well known to the users, who knew how to avoid them. However, they affected their confidence in the system.

The type ahead buffer caused problems, especially with novice users. The system displayed all intermediate states of the system, but when it was slow in responding users often press return repeatedly, or re-entered a command without realising that the system employed a buffer. As a result, tasks were often repeated and users accidentally selected incorrect functions.

The system rarely gave positive responses after completing a task, informing the user that it was completed. Instead, it returned the screen to the state it was in before the task was instantiated, i.e. the command prompt.

Sometimes, pressing the return key meant 'Yes' at a prompt, but at other times it meant 'No'. This inconsistency often caused serious problems if users were not aware of it.

With only three levels, the system functional structure was broad and simple. Users said that they did not have problems getting lost in this structure, however they wanted the facility to move sideways between the three levels. They would also have liked to use the results of one action elsewhere in the system, without having to maintain temporary Pointer Files.

From the main menu there were four options, and selecting one moved the user to the relevant part of the system. Once there, a line of information confirmed which sub-system they were in, and they were then presented with a command line. Because some of the options supported similar commands, users were sometimes confused about what part of the system they were in, and had to enter a specific query command to find out.

The system did not prevent users from performing actions which could damage the database, e.g. deletion. As a result, users were frightened to explore the system. Instead, they used the system in the way that they were rote taught by their supervisors.

There were some problems which were due to insufficient specialised cataloguing skills. The cataloguing sub-system expected the use of standardised punctuation, and abbreviation. Because of its power, some non-cataloguing staff were using this sub-system and were having problems remembering and using its specialised syntax.

Problems with integrity rules occurred when a full screen of information was displayed to the user (consisting of field names, and boxes for data entry), to enable them to enquire, add, delete or modify database information. Certain field values had integrity constraints such as alphabetic, and numeric restrictions. Some fields were also interrelated, so that the value entered in one field affected the integrity rules placed on further fields. These integrity rules had to be learnt by users, as the system gave no guidance to the rules which applied to each field. When integrity errors occurred, the system simply highlighted the rogue fields, and reported that an input error had occurred.

Users requested a facility to terminate functions once they were instantiated. They also requested some positive system response while it was performing lengthy tasks. Whenever the system failed to respond for a long period of time, users thought that either the computer had not understood what they wanted, or they had made a mistake. As a result they typed in other commands, or pressed the return key repeatedly, forgetting the type ahead buffer. This again caused problems when the system finally returned, only to process the extra erroneous commands.

Users said that concentration time was a problem, and that after a period of constant use, typically four hours, they lost concentration and often failed to notice errors in the information that they had typed. Management were trying to resolve this problem by changing working practices, and thereby reducing the stress placed upon the users.

Whenever a user made an error, the system not only informed them of the error, but usually returned them to some previous system state. This often resulted in information being lost, necessitating re-typing of the relevant data.

At the command line, the only correction key that worked was <backspace>. Although the cursor arrow keys actually moved the cursor backwards and forwards, any corrections made using these keys were not accepted by the system. However, the cursor keys functioned properly when specifying a range of database records using the example record screen. This interface inconsistency again caused some problems, which expert users eventually learned to avoid.

Another inconsistency was caused by capital letters. In one part of the system the computer distinguished between upper and lower case characters, generating an error if the user typed uppercase characters in a command. However in another part of the system, the user was required to type in uppercase characters. This caused confusion, especially when the shift lock key was utilised, and users forgot to take the lock off.

Some users expressed the desire to change the format of the field layout on data screens. These included :-

- re-ordering of fields so that they made more sense in terms of ordering or topic
- re-ordering of fields so that data input was easier, i.e. entering single columns of data in identical format, rather than using two columns
- reducing the amount of information on one screen
- dividing the screen into sections containing information that could and could not be altered.

Some users however, preferred not to change any data screen formats. They accepted what they were given, and adjusted their work accordingly.

All users described problems with memorising command syntax, and often forgot how to use certain commands over a period of time.

Once users had adjusted to the style of the system manual, they used it only as a technical reference guide whenever they forgot the exact syntax of a command.

When asked whether they would like the facility to edit and paint their own information screens, one expert user raised the following issue. If such an adaptable component was only available to expert users (because novice users may get lost, or make mistakes), then it would have a reduced effect. This is because by the time users become expert with a system, they have accepted the system standard screen layouts, and adapted themselves to any abnormalities or inconsistencies. For an expert user to then tailor the system to themselves, would entail extra work, and render some of their accumulated knowledge inappropriate. This would mean extra work modifying the system, and adjusting to its new style. Whether this would render an adaptive system useless remains to be seen. and would depend upon when adaption takes place.

4.2.4. Summary.

The library system did in fact meet the information system requirements for the library staff. However, the computer system was difficult to use and was not accepted by the majority of users. It appeared that the system design and implementation process, and the user interface software were major contributory factors to this rejection.

The major software problems identified were inconsistency, poor help and documentation, lack of integration, difficulties with modes and badly designed application functionality. These criticisms can be related to the work presented in chapters two and three. The library computer system was functionally simple, and it is probable that it would not benefit from the application of Artificial Intelligence to its user interface. Instead, attention should be given to the more fundamental software concerns listed in chapter two. In particular any software redesign should focus upon consistency within the system. Other non-software issues were also identified.

Results suggest that care must be taken during systems design, especially when end users become involved in, or are aware of the design approach taken. One point of importance arose concerning user consultation. Certain users felt that although they were consulted about their requirements and preferences, their opinions were ignored, and the resulting system did not reflect any of their comments. User consultation should only be undertaken, if the resulting comments are to be properly considered [Bennett, J.L:1987], [Eason, K.D:1987], [Galer, M:1987], [Hirscheim, R:1988], [HUFIT:Overview], [Mumford, E:1979], [Mumford, E:1981], [Poulson, D.F:1987], [Tyldesly, D.A:1987].

It was difficult to define whether the library system was or was not 'user friendly'. It became apparent that the user interface was not the only factor to affect the acceptance of a computer system. Results suggest that it is possible for an interface to be 'user friendly', while the complete computer system is still rejected by users. Likewise it is possible for a system with a 'user unfriendly' interface to be acceptable. If a system provides major improvements over existing information systems, in terms of availability and power of manipulation, then users are prepared to learn how to use a difficult interface in order to access the system. However, it is also possible for users to reject a 'user friendly' computer system because of personal issues and motivations [Green, E:1990].

Certain reasons for rejecting or accepting computer systems were identified as a result of this investigation. These are outside the sphere of influence of software or hardware engineers and are listed :-

- future job prospects
- salary benefits
- reduction in working hours
- increased power and control provided by better information
- personal objections
- dislike of stereotyped 'computer boffin' personnel
- dislike of jargon
- lack of variation in job tasks
- threats posed by the capabilities of computers
- constant workload as opposed to a workload which varies cyclically
- physical strain on eyes and body.

Unfortunately, the existing library system was already branded by its users as difficult to use, and 'unfriendly'. Further software redesign and modification would probably alleviate some of the contributory factors described above. However, an improved system may still be rejected by the users based upon experience with the existing system. The question of how to introduce improved computer systems to an already distrusting and hostile user group is outside the scope of this research.

4.3. The Influence of Systems Analysis and Design upon User Acceptance of Computer Systems.

There are many reasons why interactive computer systems fail and are rejected as 'user unfriendly' [Thimbleby, H:1983]. In particular, the library investigation highlighted the following problems :-

- designers design for themselves and not the user
- changing requirements as design develops
- users adapt to systems which are difficult to use
- confusion between functionality and ease of use
- interface design is not systematic, and it is hard to reason about design except by hindsight
- interface design is experimentally based
- designers (both application and interface) and programmers tend to have a high threshold for complexity. This makes it difficult for them to appreciate the different needs of ordinary users.

The main source of these problems appears to be a breakdown in communication between the system designer and the user [Bullinger, H.J:1987]. As a result, the designer builds systems which they think the user wants, rather than systems which they actually need and can use.

User acceptance may be considered as an equivalent system design goal to production quality control. As such, it must be made a primary objective for computer system design and implementation. More emphasis must be given to designing computer systems for the user [Gould, J.D:1987]. Furthermore, the user must not only be involved in the design process, but must be seen to play an active and influential role. User involvement should not be a front for management manipulation, it must have a genuine effect on the final system. The information processing

requirements of an organisation can be met by correct systematic information system design methods, while user interface needs can only be met by proper user consultation.

There is a need for user oriented design principles which can guide the designer in their difficult task [Thimbleby, H:1983]. These principles may be in written form, should be explicitly defined and consistently applied. Feedback from users of the final computer system should also be used to refine and extend these principles. Some example principles include:-

- the system can be used with your eyes shut
- what you can see is what you have got now
- what you type is what you get
- you can always do the same things at any time
- how you do something does not affect what you do
- you always know where you are, and what you are doing.

Design principles should be agreed by users. As discussed in chapter three, mismatches between the user's model of the system, and the actual System or Application model are a major source of interaction errors which occur during the use of a system. Matching the actual Application Model to the user's expected model is therefore the primary role of the system designer and programmer. These two roles must work in unison to both improve system design, and correct user misunderstandings. The sooner the User and Application Models are harmonised during a systems design life cycle, the better. It follows that if the user is allowed to help define the design principles, unification can begin at the start of the design process. As a result, users should be able to understand the rationale behind the Application Model, and can then begin to formulate their User Model based upon correct foundations. Subsequent system design should then match the expectations of the user.

User design principles should be standardised and compiled as design reference documents [Thimbleby, H:1986]. With regards to the library system, users held the opinion that the system was designed ad hoc and inconsistently. Explicit design standard documents agreed by users, management and system designers should not only improve the final system acceptance, but should give users and management more confidence in the design process. They should also provide a metric by which to measure the delivered system in terms of quality assurance, and software

certification. Ultimately, it should be possible to implement executable or compilable Human Factor design guide-lines, from which the final user interface is automatically generated [Galer, M:1987].

Every system is a potential experimental prototype. Even after a system is released into the commercial market place, software developers should actively encourage feedback from their customers. This feedback in regards to the 'usability' of a computer system is beneficial for both further system refinements, and the development of other independent systems.

User acceptance cannot be achieved without user exposure [Eason, K: 1988]. The amount of user exposure, and system evaluation depends upon the design approach taken. Typically there are two major approaches, the traditional 'Waterfall' System Life Cycle Model [Parkin, A:1980] and Experimental Prototyping [Gimnich, R:1987], [Hekmatpour, S:1987]. The Traditional System Life Cycle Model is based on a sequence of clearly defined Systems Analysis and Design stages. Each stage provides input to the next stage, and this input may take the form of written documentation, specifications, or program code. Individual stages may be repeated until their objectives are met and if necessary, part of the cycle may itself be repeated. Actual program implementation and testing is performed at the lower levels, only after the Systems Analysis and Design is correctly completed. Various examples of this approach include SSADM [Downs, E:1987], [Cutts, G:1987], and Jackson Structured Design [Cameron, J.R:1986]. User consultation occurs during the early stages of the system life cycle, in order to determine the system requirements. However, the user does not see the results of this consultation until the final stages, when it is often difficult to modify the design of a program. As a result, the final system may be difficult to use. Other problems also exist [McCracken, D.D:1982], highlighting the need for new, or modified approaches to system design.

Alternatively, Experimental Prototyping is based on the cycle of computer system design and implementation, followed by evaluation [Harris, J.R:1987]. A new system is then designed and implemented, and the evaluation repeated. The process continues until the designer is satisfied that a suitable system is realised. In most cases the process is repeated until no more time is left, and the last prototype may become the final system.

The evaluation cycle provides user exposure, and must be carefully controlled. If the evaluation results are to be formally used to re-specify the

next prototype, proper empirical and statistical experimental control techniques must be enforced [Jagodzinski, A.P:1988], [Monk, A:1986c], [Monk, A:1986d], [Morris, D:1988].

The potential advantages of prototyping are numerous [Harker, S:1988], [Vonk, R:1990]. Prototyping particularly favours good interface design, and ultimately a high probability of 'user acceptance' of the final system. Users actively participate and evaluate various prototype computer systems, and may observe the final system being developed. They also learn about the final system before its full implementation, and the learning is spread out over a longer period of time. Prototyping also has its problems. Prototyping is easier to use when developing small systems for small user groups [Budde, R:1984]. Inconsistencies may result, where a larger system is being developed by a team of designers for a large user population. In an attempt to satisfy the personal requirements of an individual user, a certain part of the computer system may be modified, rendering it inconsistent with other parts. Strict standards and communication controls are necessary if this effect is to be minimised. Prototyping may also take a long time to complete because of the large amount of communication and evaluation overheads. This may lead to deadlines being missed.

The use of Prototyping for successive system design may lead to the repetition of certain evaluation tasks. For example, an identical group of users should typically have the same user interface requirements for different applications. In order to 'streamline' the design process, results from previous prototype design cycles should be reusable. This may be achieved by the continued development of written design guide-lines, which may be used in future prototyping exercises. Alternatively, a specialised design Tool-set such as a UIMS may be used to carry this knowledge forward to future design projects.

The design of a fixed interface using any design approach is extremely difficult, as the question arises as to how to solve individual user differences. That is, where a user prefers an interface feature which conflicts with another feature preferred by a second user. The resulting 'best fit' approach to interface design becomes wrought with complex design decisions. Fortunately, this can be resolved using adaptive or customisable interfaces which make use of the concept of separation. These tools remove the need for some of these compromise design decisions, by providing sets of alternative interface styles.

A UIMS Tool-set based on interface separation could also be used to improve traditional life cycle design approaches. As the library system illustrated, these traditional approaches often result in computer systems which are functionally correct, but difficult to use. Modifications could be made to the life cycle, enabling UIMS interface design tools to be used in parallel with the system design stages. These tools could then be used during interactive user sessions to prototype the user interface, before the final computer system is implemented. A UIMS Tool-set should also breed familiarity. Repeated system design using the same methods and Tool-set, should help the user become more acquainted with the design process. This familiarity should both speed up and improve a design methodology.

The library system investigation revealed that proper evaluation of users using computer systems requires considerable resources and skills. In order to determine these requirements, a visit was made to the Human Factors Laboratory at ICL, Bracknell. This visit established the need for specialised and expensive recording, time stamping and play-back / mixing equipment. It also identified the necessity of a controlled work environment where different user sessions could be recorded. Finally, the visit highlighted the necessity for specially trained personnel to operate the equipment and evaluate any results produced. To illustrate the complexities involved with this type of detailed analysis, it is worth noting that one hour of video and audio tape often took the entire laboratory staff many weeks to analyse. The visit served to emphasise the difficulties of detailed computer system user evaluation, and also provided an insight into the role of the Cognitive Psychology within the field of Human Computer Interaction.

Cognitive Psychologists examine computer systems, and particularly the user interface, from the human psychology viewpoint [Dillon, A:1987]. They attempt to look deeper than the 'surface mechanics' involved in using a computer system, and seek to develop general models which describe how users actually view and use particular systems [Rasmussen, J:1987]. These models are based upon established human psychology principles such as Long and Short Term Memory, and planning strategies. Cognitive Psychologists are not directly concerned with identifying specific hardware and software problems. They are more interested in developing, testing, and refining high level theories as to how users perceive and interact with computer systems. It is left to software and hardware designers to consider these theories, or models when designing their computer systems.

Typically, this type of evaluation is far too extensive for inclusion in the regular system design process. Its main purpose is fine detailed analysis of existing computer systems. Results can then be collected together and presented as recommendations for future computer system development. The cognitive approach stands alone as a valid technique for improving the usability of interfaces. As such, work is needed to examine new methodologies for applying its accumulated research results to the interface design process.

Utilising interface separation, the results of Cognitive Psychology research can be used to develop individual UIMS components. Existing components can then be evaluated and improved, and new ones developed. Intelligent Interfaces incorporating results from this work should then closely match user requirements. This technique can also be used to identify individual user differences, which should be supported within customisable or adaptive Intelligent Interfaces.

The library system showed that many users find it difficult to differentiate between problems which are caused by them, and problems which are caused by poor design. A typical response from a non-computer expert (not necessarily an application novice) to an error, is that they are to blame. They see the Application Model as correct, and are quick to condemn themselves as being at fault. This is in stark contrast to computer experts who more confidently identify the problem as being within the computer. It is suspected that this is due to a wide range of social and psychological influences [Grudin, J:1987]. These include :-

- incorrect stereotyped views concerning the exaggerated capabilities of computers
- fear of the unknown
- personal insecurities.

These issues need to be addressed within the relevant spheres of research, with the design, introduction and acceptance of computer systems being examined within the larger context of other social, psychological and organisational influences.

Interface analysis and design, although part of the complete Systems Analysis and Design process, is separate and requires different skills.

Similarities can be drawn between both, especially concerning proper user consultation. However, these differences need addressing with different staff training and software support tools such as separable UIMSs.

4.3.1. Summary.

Experiences with the initial stages of the library system investigation identified the complexities involved, and specialist skills required for detailed user interface investigation. These experiences also helped identify the need for new software architectures which support the use of interface guide-lines and recommendations which would result from this type of experimental work. Having recognised these issues at an early stage, research emphasis was moved away from empirical investigation. Instead, effort was directed towards the design of new separable software architectures, which were identified as being fundamental if the results from this type of work are to be beneficial to user interface design.

The field of Human Computer Interaction is multi-disciplinary with the main research fields being Cognitive Psychology, Software Engineering, and Systems Analysis. There is also an overlap with other disciplines such as Social and Organisational Studies. A need is identified for new tools and methods which encompass these disciplines. Awareness is already growing concerning the influences affecting user acceptance of computer systems, especially regarding the importance of designing 'usable' user interfaces. However, existing tools and design methods can be further improved.

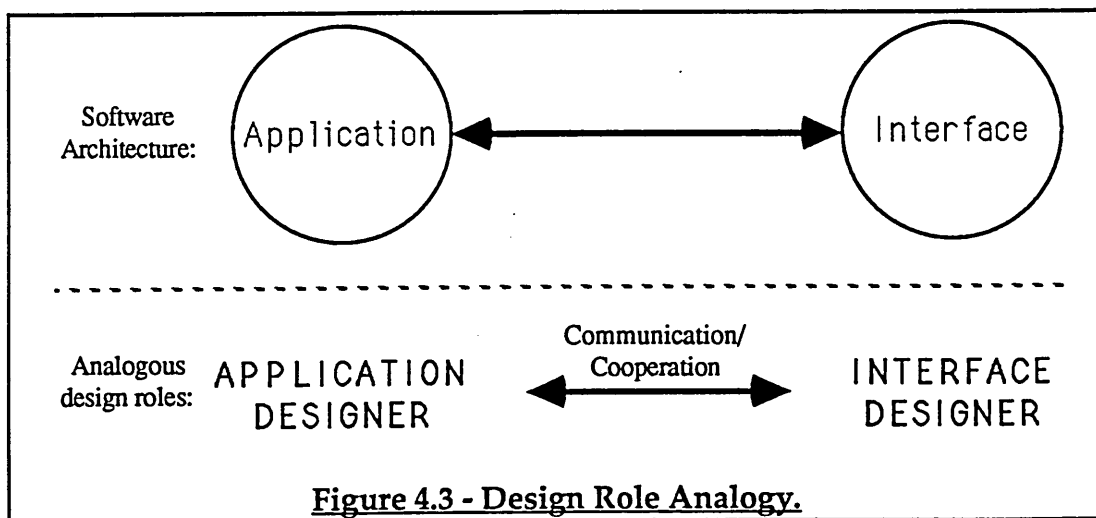


Figure 4.3 - Design Role Analogy.

Interface separation provides both a focal point for the Human Computer Interaction research community, and a meeting place for the various specialised analysis and design roles. The distinction between interface and

application facilitates the utilisation of results generated by the Human Computer Interaction research community. An analogy can be made with the separate roles of interface and application designer. The application designer may use formal methods and design tools, according to current knowledge embodied within their discipline. Similarly, the interface designer is able to apply the current interface design and implementation techniques. As figure 4.3 illustrates, these two roles can be brought together by the use of new separable software architectures. Communication between the two roles, eventually leads to the specification of the knowledge which links the user interface to the application.

An Investigation into the Quantitative User Modelling of User Interactions for the purpose of Predicting User Expertise.

5.1. Introduction.

This chapter describes an investigation into the requirements and complexities of Quantitative User modelling [Spall, R.P:1990]. An object oriented Quantitative User Model is proposed, and experiences are discussed concerning its implementation and practical application. This Quantitative User Model serves as a research vehicle, demonstrating the intricacies of the general User Modelling research field. In particular, it shows the difficulties surrounding the elicitation of useful user interaction knowledge, the structuring of such knowledge into a computable User Model, and the application of User Models as a means for inferencing user characteristics.

A Quantitative User Model embodies knowledge concerning the actual performance of a user with a particular application. Once stored in a computable form, it can be utilised to infer user characteristics such as level of expertise, and preferred interaction style. This can ultimately be used to improve the usability of a user interface. For example, in tailoring a help system to the experience level of a particular user.

Various user characteristics may be modelled in order to provide useful knowledge for an Intelligent Interface [Gilbert, G.N:1987]. User preferences can be modelled as a means of describing the personal tastes of a user, for example, preferring menu style interaction to command language, or certain text styles and colours. The modelling of user motivations and objectives could provide knowledge concerning the tasks which a user is trying to achieve. The knowledge can then be exploited within Intelligent Planning modules. The modelling of user expertise may provide knowledge concerning the understanding which a user has of the computer system being used. This can be utilised to control the amount of user assistance given by an Intelligent Interface. Similarly, modelling of user interaction skills such as mouse movement, and keyboard familiarity, should also prove beneficial, especially where disabled users are concerned. Psychological, or Behavioural Modelling could provide helpful knowledge concerning the overall characteristics and profile of a user [Benyon, D:1988]. This can then be used to match specific interface interaction styles for

individual users. Finally, the modelling of the background of a user with other computer systems should enable an Intelligent Interface to make analogies with other computer systems, and possibly adapt itself to mimic them.

A computable Quantitative User Model may use several techniques in order to determine the knowledge it contains. Stereotyping can provide an initial Quantitative User Model based upon pre-described user groups [Rivers, R:1989]. A particular stereotype is chosen from a set of general user stereotypes, according to information supplied by a user in reply to an initial question and answer session. Alternatively, the appropriate stereotype may be directly specified by the user, interface designer, or automatically selected based upon initial user interactions [Benyon, D:1987]. This approach limits the flexibility of a Quantitative User Model, and assumes that it is possible to classify users into groups according to psychological or behavioural frameworks. Instead, a Quantitative User Model may determine its knowledge using an extensive question and answer session with the user [Murray, D.M:1987]. This approach makes use of well proven psychological and behavioural questionnaires and analysis techniques, and a finer granularity is achieved. However, this technique is obtrusive and requires that the user answers questions accurately. It also assumes that users can describe themselves correctly, and that the analysis techniques work. Finally a Quantitative User Model may derive its knowledge from actual user interactions [Warren, C:1987]. This requires that the Quantitative User Model monitors user interaction and from this infers knowledge about the characteristics of a user. The Quantitative User Model must be capable of inferring the intentions of a user and recognising any difficulties, contradictions, or conflicts which occur. This technique is relatively unobtrusive, but problems may arise due to incorrect inferencing. Some Quantitative User Models may in fact combine these techniques. For example, using stereotyping to determine the initial Quantitative User Model knowledge, while employing interaction inferencing techniques to refine its accuracy.

Several types of information concerning user interactions are available to a Quantitative User Model for inferencing purposes [Corbett, M:1987]. These include correct application usage, incorrect usage or errors, requests for help and tutorial support, and time lapses between usage. These interactions can each be used to infer different types of knowledge concerning the user. They can also be used to infer different things at different times, depending upon

the context of their occurrence. For example, correct application usage can be used to infer user development, that is progression towards an expert user. However, application errors can be used to infer user regression at certain times, while at other times they may be used to infer user progression. This may occur when an expert user is exploring a new application, or is using negative error responses to interrogate an information system. Inferencing based upon user interaction is also dependent upon the learning abilities of a user, which must therefore also be represented within a Quantitative User Model.

Three main issues therefore need addressing within the field of Quantitative User Modelling, namely knowledge representation, knowledge acquisition, and knowledge inferencing [Payne, S.J:1987]. Knowledge representation deals with the actual structure of a Quantitative User Model. That is, what knowledge it contains and how the knowledge is actually structured. The structure can be separated from the knowledge itself, and may therefore be used to structure many similar types of Quantitative User Model. Internal structure is important, and ultimately determines the usefulness of any Quantitative User Modelling technique. Knowledge acquisition addresses the problem of determining the knowledge contained within a Quantitative User Model. In certain situations this may be a 'one off' process in order to define the initial Quantitative User Model. However, it is typically a continuous process, as the user being modelled is constantly changing. Finally, knowledge inferencing is concerned with the way in which a Quantitative User Model is used to infer new knowledge. This knowledge is not new in terms of being non-deterministically created by the Quantitative User Model, rather it arises from the structure of the Quantitative User Model and the intelligent heuristics which make use of this structure. This illustrates a major use of modelling, which is to represent existing knowledge using new structures, in order to provide different ways of looking at a problem. In doing so, further knowledge may be derived which only exists as a result of combining existing fragmented and disparate knowledge.

This proposed Quantitative User Model tests the hypotheses that the interactions of a user with an application, can provide the basis for predictive reasoning to determine knowledge concerning the expertise of a user. Such knowledge can then be used within an Intelligent Interface. For example, an Intelligent Help System may be derived for the purpose of tailoring its response to user requests for assistance. The proposed

Quantitative User Model was developed as a 'test vehicle', and uses a mathematical approach to describing knowledge concerning a user. It embodies knowledge concerning correct and incorrect application usage, and takes full advantage of the principles and benefits of object oriented design. User expertise is classified as either Novice, Intermediate, or Expert, and classification is applied to three object oriented conceptual levels, namely application, object and function. However, the Quantitative User Model does not extend to error recognition and classification, and it is assumed that this knowledge will be supplied by the user interface. The Quantitative User Model represents the learning abilities of a user with a traditional Learning Curve.

5.2. Overview of the Proposed Quantitative User Model.

The expertise of a user with a computer application can be divided into different conceptual levels. For example, overall interface expertise, overall application expertise, and both individual interface and application component expertise. The proposed Quantitative User Model only considers user expertise with an application, and organises this according to three conceptual levels. These are overall application expertise, expertise with component objects, and expertise with object functions.

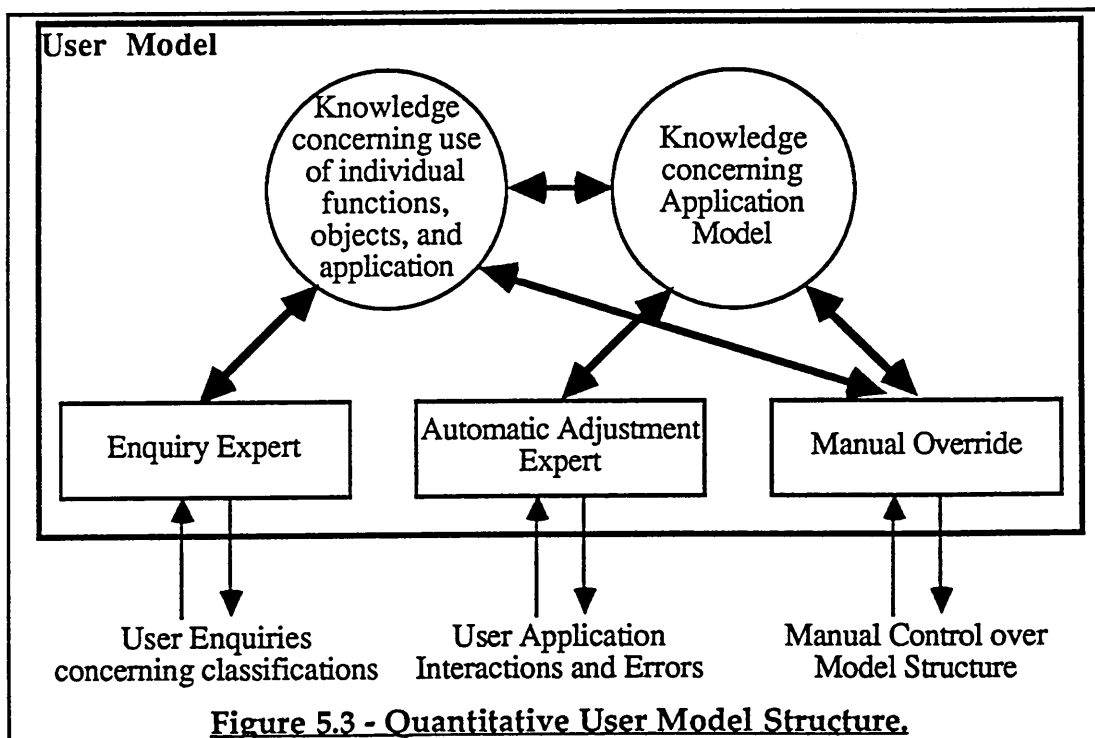
Individual user expertise classification is required for the various objects and functions at different conceptual levels. This classification may be stored directly as a value, or inferred from other knowledge contained within the model. Classification is maintained for individual components at each conceptual level, rather than as a general level classification. For example, users are classified accordingly for each individual object, application and function rather than being given a single classification which generally applies to all applications, objects or functions. Usage of functions and objects at different conceptual levels indirectly affects the expertise of a user with other functions and objects. For example, individual function usage may well affect the overall expertise of a user with an application. Similarly, use of one function may affect the expertise of a user with other related functions. This shows the need for an explicit Application Model which describes the relationship between various application functions, and the objects on which they act.

Traditional procedural languages do not provide this self describing function, and the application architecture is often hidden within the

program structure. Because of this, separate Application Models are required which must be specified in computable formats [Carberry, S:1988]. These can then be extended to incorporate knowledge which describes the relationships between individual functions and conceptual objects. Fortunately, object oriented languages support a self describing explicit Application Model based upon an application, its component objects, and their behaviour (which is determined by their explicit message protocol). This is utilised by the proposed Quantitative User Model.

5.3. Structure of the Proposed Quantitative User Model.

Figure 5.3 illustrates the basic Quantitative User Model architecture, and the information flows between the application and the Quantitative User Model. An object oriented application is used by sending messages between different self contained objects. The architecture uses this feature to determine what objects and functions were correctly or incorrectly used. The Quantitative User Model is implemented separate from the Smalltalk message handling mechanism. It therefore expects to be independently informed of the correct and incorrect messages issued by the user, and those arising from communication between various application objects.



As shown in figure 5.3, the Quantitative User Model is divided into knowledge and heuristics, which are shown using circles and boxes respectively. Knowledge is implemented using internal data which is

hidden inside the Quantitative User Model, and describes both the use of an application by a user, and the Application Model. The knowledge concerning application usage is related to the Application Model, and is therefore structured accordingly. Separate heuristics, implemented as Smalltalk 80 code, access and modify this knowledge. These heuristics are divided into three groups, and provide an interface between the Quantitative User Model and the user.

The Application Model is represented at three conceptual levels. These are based upon the object oriented concept of an application comprised of objects which provide a specific function, or method protocol. The three levels are :-

- application
- application objects
- object methods or functions.

The application effectively represents the highest conceptual level, while object methods denote the lowest. There may also be many objects and methods at the intermediate object and method conceptual levels. These are referred to as Application Components, and are individually represented within the internal Application Model. The relationship between the individual Application Components belonging to an application are also contained within the internal Application Model. The object to which a method belongs must always be defined. Likewise, the application to which an object belongs must also be specified.

The following information is stored within each Quantitative User Model, for each Application Component :-

- number of uses
- date of last use
- number of errors (for each error type)
- number of uses since last error (regardless of error type)
- Intermediate Classification Trigger (controls move from Novice to Intermediate)
- Expert Classification Trigger (controls move from Intermediate to Expert)
- current Experience Classification (when set to nil, user classification is inferred from knowledge contained within the Quantitative User Model. When set to a numeric value

between 1 and 3, it is assumed that the automatic inferencing is to be overridden with a classification specified by the user. A value of 1 implies Novice experience, while 2 and 3 imply Intermediary and Expert experience classifications)

- Usage Transfer Factor, i.e. the effect of usage on higher conceptual level (not maintained at application conceptual level)
- a single mathematical formula is also maintained for each Quantitative User Model to represent the Learning Curve for a user.

This knowledge is constantly updated according to the interactions made by a user, and is maintained between different sessions. The purpose of this knowledge is described below in section 5.4.

The complete Quantitative User Model is implemented as a single Smalltalk Class, with the internal knowledge encapsulated within instances of the Class. This class mechanism is further discussed in section 6.2 and appendix B. The necessary heuristics are implemented as instance methods using Smalltalk code. A user requires a separate quantitative Quantitative User Model for each application, and this is achieved by creating new instances of the *QuantitativeUserModel* abstract Class. Each Quantitative User Model effectively contains separate knowledge, but shares the same instance methods or heuristics defined by the *QuantitativeUserModel* abstract Class.

Individual application objects are implemented using the object oriented Class mechanism [Goldberg, A:1981]. Application objects which share the same Class definition behave identically, and knowledge concerning their usage need only be modelled at the Class level. Multiple instances of the same Class are therefore represented as a single Application Component within the Quantitative User Model, and usage of individual instances only affects the one Quantitative User Model Class Application Component representation.

Objects sharing the same Class definition may also be used within different applications. The objects used within a specific application are not explicitly described, and the Quantitative User Model needs to be told which Class definitions relate to which application. This knowledge is incorporated within the internal Application Model. The first step in using the Quantitative User Model must therefore be to define the Classes used by an

application, and also the messages which any Class instance, or object can understand. This is achieved by sending the appropriate messages to the Quantitative User Model.

5.4. Functioning of the Proposed Quantitative User Model.

Three function or method sub-sets are provided by the Quantitative User Model. The first 'Automatic Adjustment Expert' sub-set enables the Quantitative User Model to be informed of application usage. It contains the necessary heuristics to modify the Quantitative User Model according to specific user interactions. The second 'Enquiry Expert' sub-set provides inferencing heuristics for classifying the expertise of a user. Finally, a third 'Manual Override' sub-set allows the internal Quantitative User Model knowledge to be manually modified, thus providing an override facility. Appendix C contains the relevant Smalltalk 80 implementation source code.

The Automatic Adjustment Expert provides a set of messages which must be used to inform the Quantitative User Model of specific application user interactions. Interactions may occur at the application, object or method conceptual levels, and must be identified using appropriate names. When object usage is reported, the application in which the object is being used must also be identified. Similarly, when method usage is reported, both the object to which the method applies or belongs, and the application in which the object is being used must be specified. This enables both correct and incorrect application usage to be monitored.

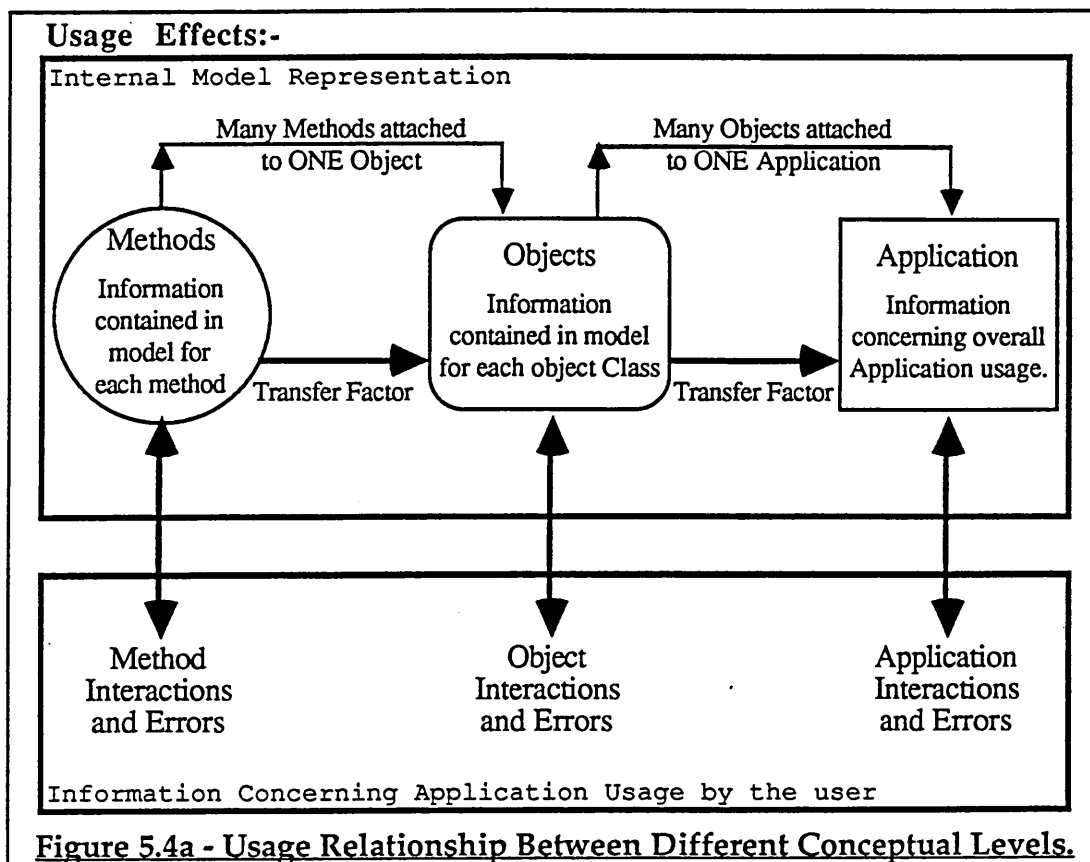
If the Quantitative User Model is informed of an interaction using an object or method which is not described within its Application Model, the new object or method is automatically added. In doing so, default values are set for the usage Transfer Factor, and Classification Triggers.

When the Quantitative User Model is informed of correct application usage, the appropriate internal Usage Count is incremented. Similarly, when incorrect usage occurs, the relevant Error Count is increased. The date since last usage and number of uses since last error are also updated accordingly.

Using a particular method not only affects the expertise of a user with that method, but also with the object to which it belongs. Similarly, use of an object also affects the expertise of a user with the application of which it is a part. The Quantitative User Model imitates this effect by providing a

Transfer Factor for each Application Component. This determines the effect of the usage of one component upon another component at a higher conceptual level. For example, the effect of the usage of a method upon the object to which it belongs. As illustrated in figure 5.4a, this Transfer Factor is a simple decimal number which is passed on to the component at the next conceptual level. In the case of method usage the next conceptual level is the object conceptual level, while in the case of object usage the next level is the application conceptual level. The Transfer Factor can be set for individual Application Components and should be in the range of 0 to 1. As an example consider an instance of the Person Class which understands the *age* message. Use of the *age* method increases the Usage Count by one, for the Person Class *age* method defined within the Quantitative User Model. If the *age* method Application Component has a Transfer Factor of 0.2, then using it causes the Usage Count for the Quantitative User Model Person Class definition to increase by 0.2, i.e. $1 * 0.2$. Assuming that the Person Class Application Component has a Transfer Factor of 0.1, the appropriate application Usage Count is finally increased by 0.02, i.e. $0.2 * 0.1$.

An application may comprise of many objects, each of which has a separate Transfer Factor. Similarly, each object may have many methods each of which also has a separate Transfer Factor. The Usage Count for an Application Component may therefore be directly affected by user interaction, or indirectly by the use of a related Application Component at a lower conceptual level.



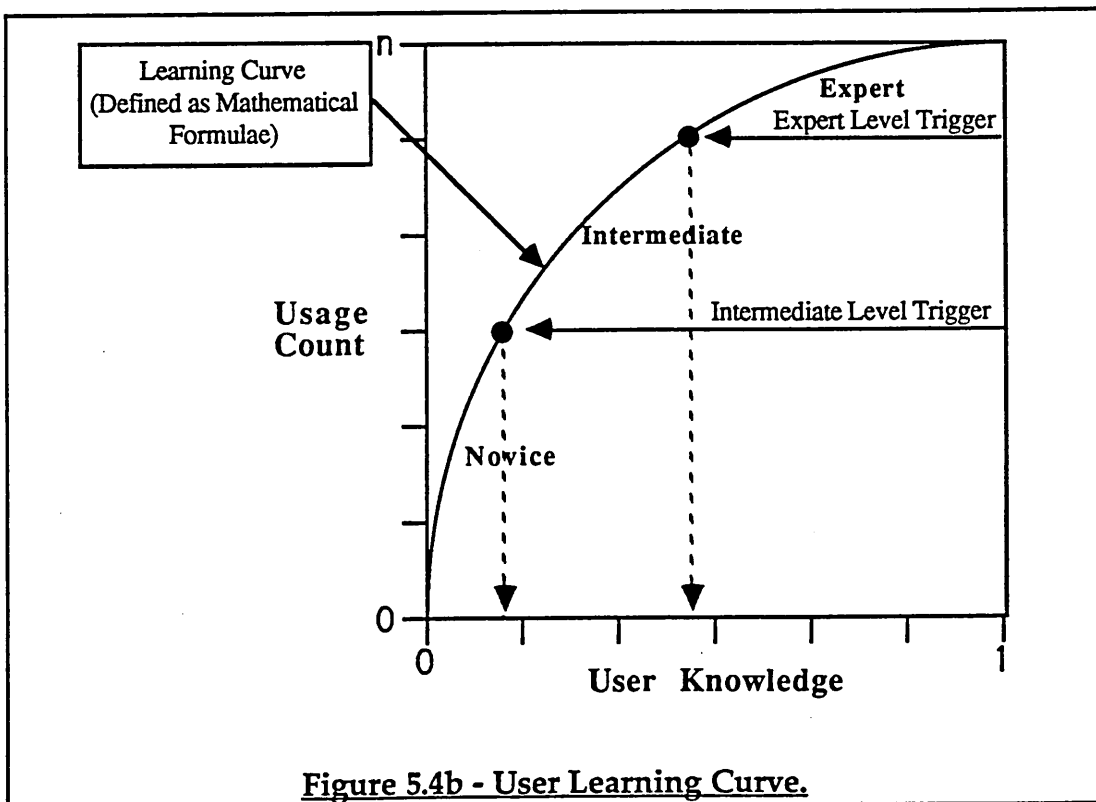
Although the Quantitative User Model does not provide any error recognition or classification facilities, it does support more than one Error Count for each component in order to represent different types of error. It is up to the implementor of the Quantitative User Model to interpret the meaning of different error types, and to inform the Quantitative User Model when errors of a particular type arise. When the Quantitative User Model is notified of an error made while using an Application Component, the appropriate Error Count is incremented by one. Knowledge is also maintained concerning Application Component use since the last error was made, and this is reset to one whenever an error of any type occurs.

Unlike correct interactions, the effect of an error is passed onto the next conceptual level on a one to one basis. This is analogous to setting the Transfer Factor to 1. For example, consider the effect of a user who makes an error of type One with a particular method. This would cause an increase of one in the type One Error Count field for that method, the Class to which it belongs, and the application in which it is being used.

A Learning Curve is maintained within the Quantitative User Model in order to represent the ability of a user to learn new knowledge concerning Application Components, and is stored as a mathematical formula. The

Learning Curve is used to define the rate at which a user learns knowledge, and assumes that the user learns all knowledge at the same rate across different conceptual levels. As figure 5.4b illustrates, this formula is used to plot a graph showing Usage Count against User Knowledge. The User Knowledge is shown along the vertical axis, and ranges from 0 (Novice) to 1 (Expert). The Usage Count is shown along the horizontal axis, and its range may differ between Quantitative User Models. The Learning Curve formula is stored in the form $X = f(Y)$, where Usage Count can be substituted for Y, and X represents actual User Knowledge.

Two triggers, the Intermediate Classification Trigger and the Expert Classification trigger, are used to calculate the classification of a user for a particular Application Component. These triggers represent the User Knowledge required for a user to move from novice to intermediate classification, and from intermediate to expert classification respectively. Their values should range from 0 to 1, and the Expert Trigger should always be greater than, or equal to the Intermediate Trigger. These triggers break the Learning Curve into three regions, namely: novice, intermediate, and expert. Using these triggers, the Learning Curve formula and the current Usage Count, the user classification for a particular Application Component is computed. Although a single Learning Curve formula is used for the entire Quantitative User Model, the two Triggers may be set for individual Application Components.



The knowledge contained in the Quantitative User Model is automatically updated according to user application interactions. However, this knowledge may also be directly modified using the Manual Override methods provided. These enable the internal Application Model structure to be changed, as well as the information stored for each Application Component - as listed in section 5.3 above. The Quantitative User Model Learning Curve may also be modified. Finally, the automated expertise classification mechanism may be overridden, and an explicit experience value set for each Application Component. This may be set to either 1, 2, or 3 implying Novice, Intermediary, or Expert knowledge with the appropriate Application Component. Automatic classification may be resumed at any time by setting this value back to nil. Again, Quantitative User Model methods are provided to simplify switching between manual and automated classification.

5.5. Evaluation and Conclusions.

After initial testing, it was realised that the proposed Quantitative User Model could not accurately predict the expertise of a user based upon correct usage alone. In response, attempts were made to modify the heuristics contained within the Quantitative User Model to take into account both errors and time lapses. Again, the Quantitative User Model failed to accurately predict the expertise of a user. The reasons for this inaccuracy appeared to lie in the complexity of structuring, acquiring, and inferencing from quantitative knowledge concerning a user.

Determining the initial Quantitative User Model knowledge was difficult. In particular, it was hard to fix the Application Component Classification Level Triggers, Learning Curve formula, and Transfer Factors. Assuming that these values could be accurately calculated, extensive empirical work would be necessary to investigate the mathematical relationship between the different values for each Quantitative User Model, and the individual user characteristics being represented. This definition stage would also take a long time as there appear to be numerous interrelationships between Application Components at different conceptual levels.

User interaction is complex, and the relationship between correct usage and user expertise is difficult, and may be impossible to mathematically express completely. Similarly, the effect of errors and time lapse is also difficult to

mathematically represent. User interactions can not be considered independently of other factors which also affect Human Computer Interaction. These include work environment, personal motivations, work pressures, and stress. Again, assuming that this extra knowledge could be acquired, the complex relationships that result would require tremendous computing power to solve.

While using the Quantitative User Model, it was realised that certain types of knowledge concerning the application bore no direct relationship to the actual Application Model. For example, knowledge describing the general understanding of information systems, or object oriented concepts by a user. Similarly, knowledge concerning user understanding of the application domain and terminology was not described. Without this knowledge it would not be possible to accurately determine user expertise with different Application Components. The knowledge represented within a Quantitative User Model would therefore have to be expanded to include information concerning these, and other more general concepts.

In effect a large amount of extra knowledge is required to improve the accuracy of the proposed Quantitative User Model. This raises the issue of organising, capturing, and using this extra knowledge within a Quantitative User Model. Reflecting upon the complexities of these issues, the implementation and use of accurate Quantitative User Models will undoubtedly require both immense computer resources and Artificial Intelligence.

Another problem identified by the proposed model relates to inconsistency between the Quantitative User Model and the real user. If small inconsistencies were not immediately corrected, the Quantitative User Model quickly became inconsistent with the real user, and therefore useless. Ultimately, this implies that a Quantitative User Model has to exactly represent the user at all times or not at all. This could be achieved by periodic question and answer sessions with the user. However, the overhead of extra question dialogue may defeat the very purpose of Quantitative User Models. Alternatively, the heuristics for updating and inferencing from the Quantitative User Model must take account of every factor which can possibly affect the real interaction processes of the user.

A final problem is that of heuristics which may themselves change. The heuristics for updating the Quantitative User Model and for inferring

knowledge about it, were themselves subject to change. For example, the effects of errors and correct usage upon user expertise changes, as a user moves from a Novice to Expert. Similarly, these effects are probably different during each cycle of the repetitive move from Novice to Expert, and Expert back to Novice over a period of time. It is also probable that the mechanisms for changing the heuristics are themselves subject to change, and so on.

5.6. Summary.

The very least that this research shows is that an object oriented design facilitates the definition and updating of a Quantitative User Model to represent user interactions with a software application. The mechanism for monitoring and describing user interactions are relatively straight forward, and can be implemented within the interface software. The body of knowledge required to define the parameters influencing the 'shape' of the Learning Curve for individual users is less clear, and needs to be extracted from the fields of Educational Psychology, Cognitive Psychology, and Artificial Intelligence. The future of User Modelling, and in particular Quantitative User Modelling depends upon the success of these fields in solving the problems identified.

An Object Oriented User Interface Management System, and Integrated Interface Design Tool-set.

6.1. Introduction.

This chapter proposes a new object oriented User Interface Management System (UIMS) implemented in Smalltalk 80. This UIMS uses a novel software architecture based upon distinct interface and application separation. It also incorporates an integrated interface design Tool-set enabling the re-use of interface components within different interfaces. This Tool-set assists the interface designer in building separable re-configurable Direct Manipulation Interfaces for object oriented applications. The proposed UIMS has arisen out of the need for new interface design methods which was outlined in chapter three, and will eventually form part of a complete integrated UIMS for generating Intelligent Interfaces.

6.2. Overview of Object Oriented Programming.

Central to object oriented programming is the concept of a self contained object. This object contains its own data representing its state, and exhibits certain behaviour according to the functions it can perform. The data belonging to an object can only be referenced by the functions defined for that object, and not directly from outside of the object. It comprises of a list of field names, each of which points to another object. Individual functions are named, and have attached code which is executed whenever they are called. They usually return a value, and may allow arguments to be included. These returned values and arguments are normally themselves objects.

An object oriented application is used by instantiating object functions, usually by sending a message to an object with a name which matches the appropriate function name. The object which receives the message is known as the Receiver. The instantiated function code may then send further messages to objects pointed to by its data fields, and so on. As an example consider the expression '1 + 2'. In object oriented terms, 1 and 2 are both objects, and + is a message or function name. The + message requires a single argument, in this case the object 2, and returns the value 3 when sent to the object 1.

The structure of every object is described by an abstract data type, which for the purpose of this research is known as a Class. Every object is an instance of a Class. One Class may have many instances, and instances of the same Class share the same function definitions. However, they contain separate internal data fields representing their individual state.

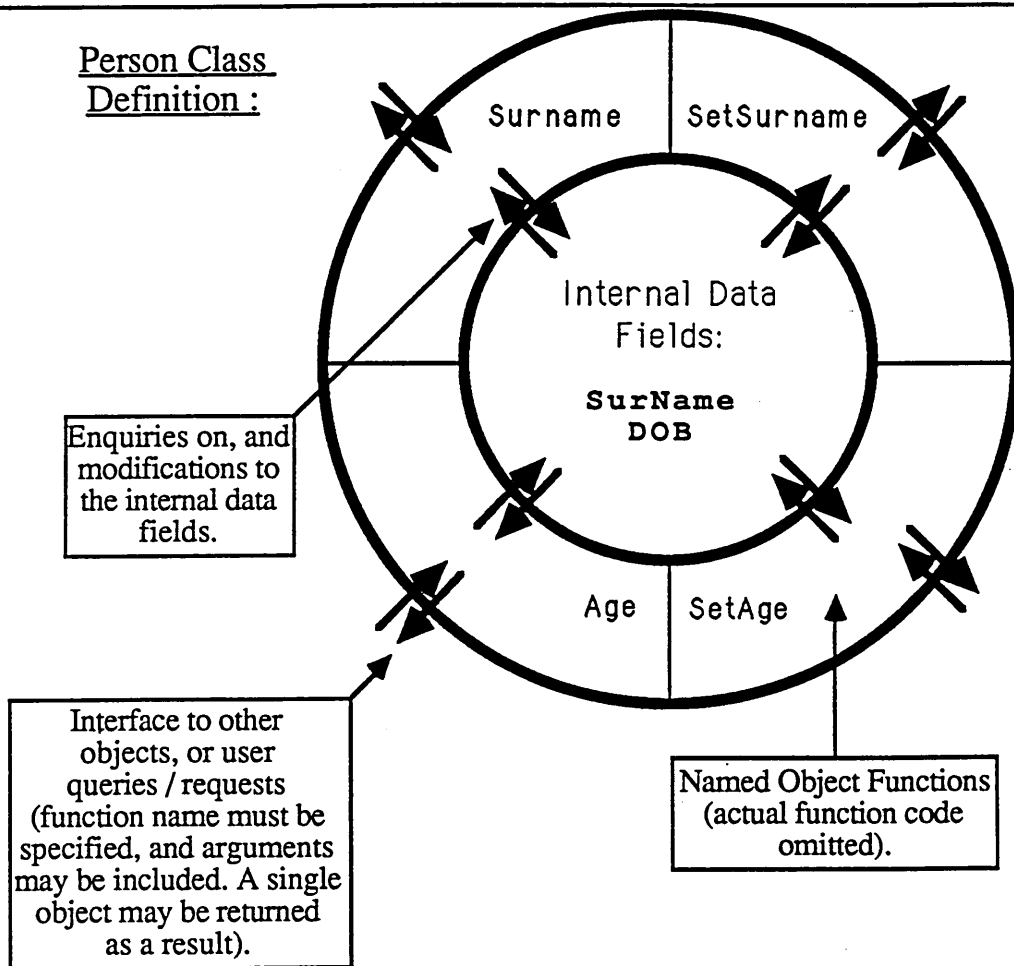
A given Class, 'A', may be the Sub-class of another Class, 'B'. In such a case, 'B' inherits the function definitions, and internal data fields defined in 'A', and may add its own functions and internal data fields. This is known as inheritance and results in an inheritance hierarchy tree (or a net if a Class can inherit from more than one Super-class).

Figure 6.2 illustrates the object concept, and the relationship between instance and Class. It shows how an object presents itself as a separate unit with a self contained state, exhibiting a certain type of behaviour. The Person Class defines two internal data fields for all of its instances, and four functions which can be used by any instance. Two of these functions serve as queries, and reply with the Age and Surname of a Person instance. The remaining two modify the state of an instance and allow the Age and Surname to be changed. As such, the two functions *SetAge*, and *SetSurname* require arguments.

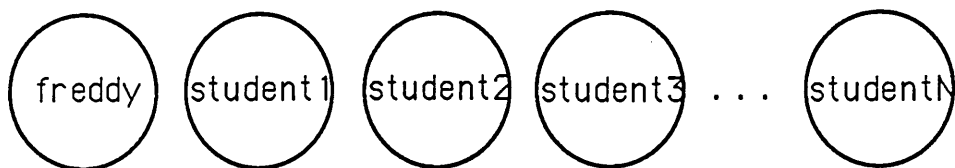
In order to design an object oriented application, the designer must decide:-

- the object Classes to include in a system
- the internal representation of their state (i.e. what data fields)
- the behaviour they exhibit (i.e. the function definitions)
- the inheritance structure between Classes
- the relationships that exist between Classes, e.g. part aggregation, associations, generalisations, and any qualified relations.

Person Class
Definition :



Named Person Class instances :-



(each has its own separate internal data fields, and shares the same function definitions)

Figure 6.2 - Object Class Definition and Instances.

When an object is presented to a user, they do not need to be aware of the internal implementation of the object, they merely see the name, or identifier of the object and the set of messages they can send to the object in order to instantiate tasks. Consider the Person Class shown in figure 6.2. To enquire on the Age of a Person, the function Age is used. The actual age is not maintained as an internal data field but is worked out from the date of birth (DOB internal data field). The user of this object or function need not be aware of this. This is known as data hiding, and provides object encapsulation [Thomas, D:1989].

Finally, object oriented languages support polymorphism. Objects may respond differently to the same message, according to their specific Class. In effect, similar objects provide identical external function names which each implement different behaviour. For example, a set of graphical objects such as rectangles, circles, and polygons, may provide the same external functions, such as *draw*, *rotate*, *translate*, *enlarge*, and *reduce*, while individual objects may implement these functions differently. Conventional imperative programming languages would require complex conditional statements to simulate polymorphism, while object oriented languages simplify its implementation, and present it to the programmer in a 'more natural' way.

A wide range of object oriented design methods are available to the software engineer [Bailin, S.C:1989], [Booch, G:1986]. Object oriented software designs may be implemented using specialised object oriented languages, or using traditional imperative and functional languages. The latter can be difficult, and requires special programming skills, and careful application of the available languages facilities. Typically, there is a move towards adding object oriented facilities to traditional languages, for example LOOPS [DeMichiel, L.G:1987], and Objective C [Rawlings, R:1989], C++ [Stroustrup, B:1986]. These hybrid languages are powerful, and can help the programmer move from traditional imperative and declarative programming concepts to those of object oriented programming. However, there is the danger of mixing traditional programming techniques with object oriented techniques. This can result in complex programs which are labelled as object oriented, but in reality are not. An alternative is to use recognised object oriented languages such as Simula [Birtwistle, G.M:1973], Ada [Barnes, J.G.P:1980], EIFFEL [Meyer, B:1987] and Smalltalk 80 [Mevel, A:1987]. Although, the ideas presented above form the basis of most of these languages, each portrays object orientedness in a slightly different way, and as such provide different concepts and structures. The Smalltalk 80 language was chosen for this project, and is briefly outlined below.

6.3. The Smalltalk 80 Programming Language and Environment.

Smalltalk 80 is an object oriented language, incorporated into an interactive environment. Everything in Smalltalk is an object (an instance of a Class), and therefore provides for a consistent and 'pure' object oriented software design. Smalltalk applications are developed by adding, and changing Class definitions. The Tool-set for doing this is also built around objects, and can

itself be modified accordingly. The complete Smalltalk 80 system is not described in any great detail, but certain elements do need attention. Appendix B contains a concise description of the salient Smalltalk features which are relevant to this UIMS implementation, and deals with object representation, creation, inter-object communication, inheritance, polymorphism, and explains the important concept of object dependency. Further Smalltalk reference material includes the Smalltalk 80 Reference Guide [Smalltalk80:ReferenceGuide], the Xerox Park Smalltalk 80 books [Goldberg, A:1983], [Goldberg, A:1983b], Mevel [Mevel, A:1987], and Byte Magazine [Byte:1981].

6.4. The User Interface Management System and Software Architecture.

The underlying UIMS software architecture divides a software application into three sub-systems; Application Functions, Interface View and Interface Interaction Controller. The Interface View and Interface Interaction Controller sub-systems represent the user interface, which is clearly separated from the Application Functions. This enables an interface to be implemented separate from the application using the Tool-set provided. The concept of a Pluggable View Controller (PVC) was conceived as a mechanism to achieve proper interface separation. The PVC mechanism provides a generic software architecture component from which complex direct manipulation interfaces can be built. As a result, the subsequent interfaces exhibit clear separation from the underlying application, and enable the advantages of distinct interface separation to be utilised.

The PVC mechanism comprises of separate View, Interaction Controller, and Separation Controller components. Together, these form the complete interface which is separated from an application using the Separation Controller. Interfaces implemented using the PVC mechanism have an underlying formal interface description which describes what the interface looks like. The proposed UIMS uses this formal syntax to construct the appropriate interface and present it to the user. The UIMS Tool-set allows individual interfaces to be interactively created using the PVC mechanism, and supports the automatic generation of the relevant underlying formal interface description. The UIMS effectively executes formal interface descriptions generated by the integrated interface design Tool-set.

In object oriented terms, the application appears as an Application Object. An Application Object may comprise of many other smaller objects, each with their self contained state. It appears to the PVC as a single object, and provides a set of Non-Interactive Functions which can be accessed by the PVC. The following implementation assumes that a single Application Object exists for each application, which provides a 'front end' through which any constituent objects are accessed.

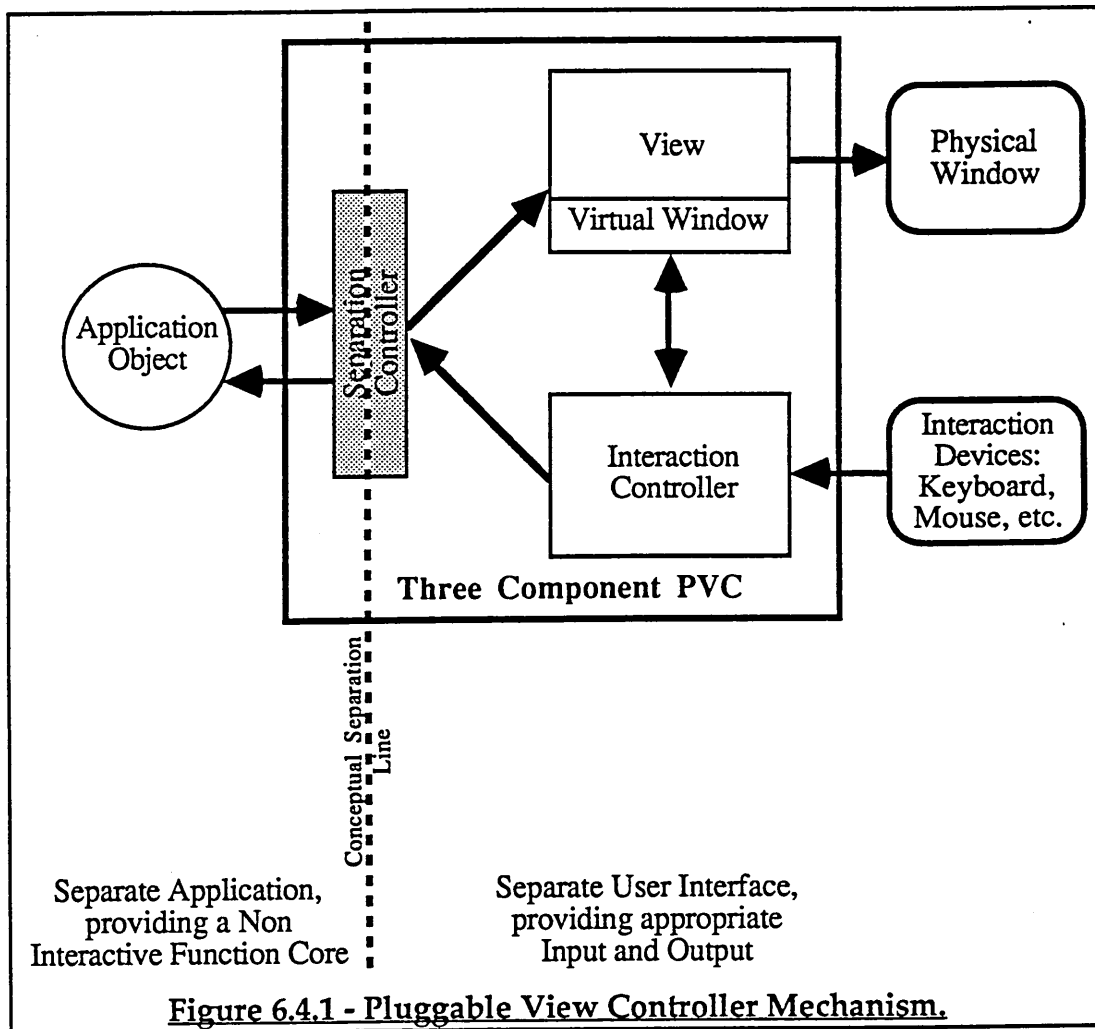
6.4.1. The Pluggable View Controller Mechanism.

Figure 6.4.1 illustrates the Pluggable View Controller mechanism. The View component controls user interface output, the Interaction Controller handles user interactions, and the Separation Controller provides application and interface separation.

The View provides a virtual window area to which graphics and text can be written. Whenever an interface is used, this virtual window is mapped onto a physical display window. The size and location of the physical window is variable, and is set by the user. The constant virtual window is then automatically scaled and translated to match the physical window.

The Interaction Controller handles user input. This component monitors the various input devices (i.e. keyboard, and mouse) and processes the relevant user interactions.

The Separation Controller controls all communication between interface and application, enabling both interface and application to be designed and implemented independently. It is used to contain knowledge which maps the interface processes on to appropriate application functions.



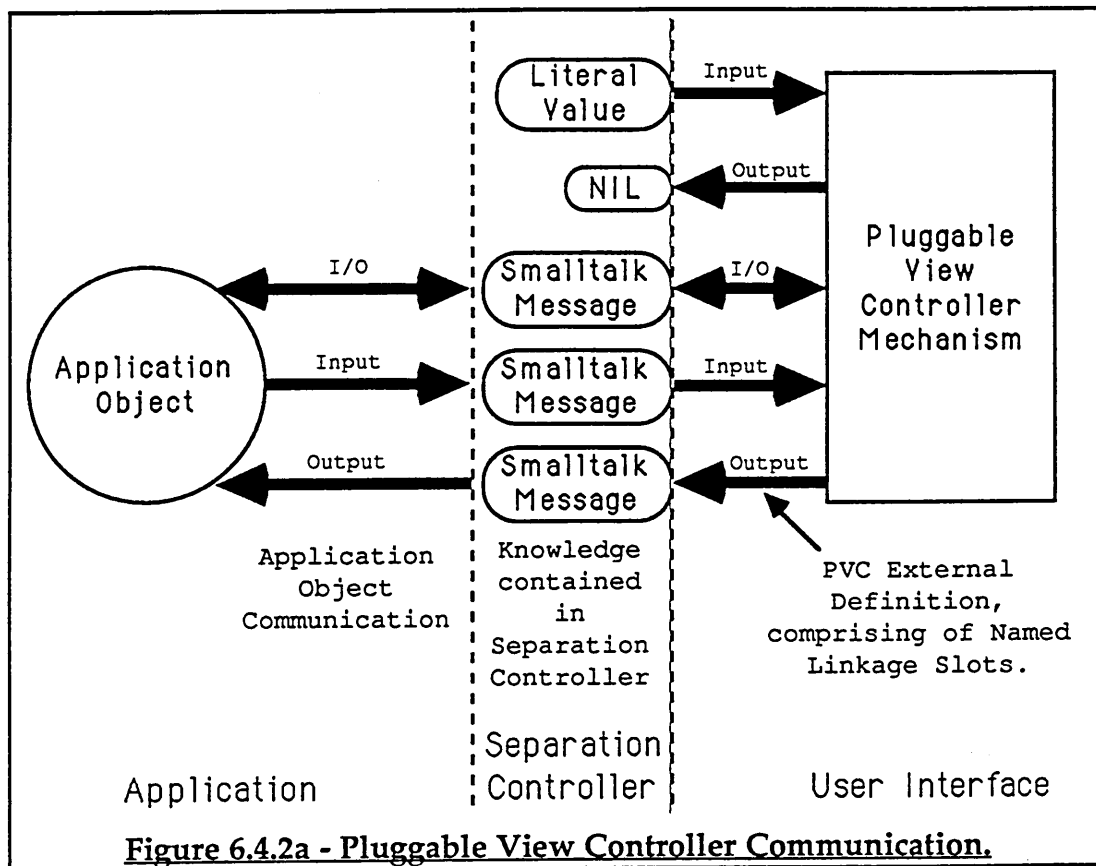
With this approach, the complete Pluggable View Controller (PVC) mechanism, can be effectively 'plugged' onto Application Objects. The state of an application is then presented to the user through the PVC View, while the Interaction Controller allows the user to interact with the Application Object. The 'plugging' process entails defining which Application Object functions map onto the Interaction Controller and View processes, and is performed by the interface designer. Any number of different interfaces can be implemented for the same Application Object, giving the user a choice of interface styles. For example, several different PVC mechanisms could be implemented to provide alternative interfaces to a personnel database Application Object which provides functions to enquire upon the hourly pay, and number of hours worked by an employee. One PVC mechanism may display these values as text upon the screen, and provide an editor with which to modify them. Another PVC may display the values using sliding bars, and allow the values to be modified by altering the position of the relevant bar. In each case, the PVC Interaction Controller would handle user interactions, while the View would handle output to the screen. The Separation Controller would then map the relevant user interaction and

display functions which require information from, or modify the application, onto appropriate Application Object functions. The same PVC mechanisms used to provide a sliding bar or text editor can also be re-used within other interfaces. Neither an Application Object, nor the interface (i.e. Interaction Controller and View) can perform this 'plugging', as the two are implemented separately and brought together by the interface designer.

Communication between the Separation Controller and Application Object, View and Application Object, and Interaction Controller and Application Object takes the form of Smalltalk message passing. This consists of service requests made to the Separation Controller. For example, an Application Object sends messages to the Separation Controller informing it of any changes made to itself. The appropriate messages are then passed onto the relevant PVC View and Interaction Controller components, according to the knowledge contained in the Separation Controller. Similarly, when the Interaction Controller informs the Separation Controller of any interactions which occur, the appropriate Application Object functions are instantiated.

6.4.2. Communication Between Objects and Pluggable View Controllers.

Communication between the interface and Application Object may only occur indirectly through the Separation Controller. This ensures that PVCs and Application Objects can be implemented separately. Indirect communication is explicitly defined for every PVC implementation, and is achieved using Linkage Slots. These Linkage Slots define what input is required from an Application Object, and what output can be sent to an Application Object in relation to the PVC. Although the View and Interaction Controller define what Linkage Slots are required, the Separation Controller contains the actual knowledge used for communicating. This effectively allows the same PVCs to be re-used by changing the knowledge contained in the Linkage Slots.



As shown in figure 6.4.2a, each Linkage Slot is identified with a unique name, and there are three types of Linkage Slots defined in relation to the PVC, namely: Input, Output, and I/O. Output Linkage Slots send results to the Application Object, or inform of interactions such as button presses, and may require arguments. Input Linkage Slots gather information from, or determine the state of the attached Application Object. They do not normally require arguments, and results are returned to the PVC as Smalltalk objects. I/O Linkage Slots are used for both Input and Output.

Each Linkage Slot must contain either a Smalltalk 80 message, a literal value, or nil. A nil value implies that a Linkage Slot is not being used, and the PVC implementation should define the appropriate default when input is required. In the case of output Linkage Slots, a nil value implies that no message is sent. A Smalltalk 80 message implies that a specific message is sent to the Application Object in order to inform of an interaction event, or to obtain an input value. A literal value may only be used for an Input Slot. Instead of sending a message to the object, the specified literal value is returned whenever input is required by the PVC.

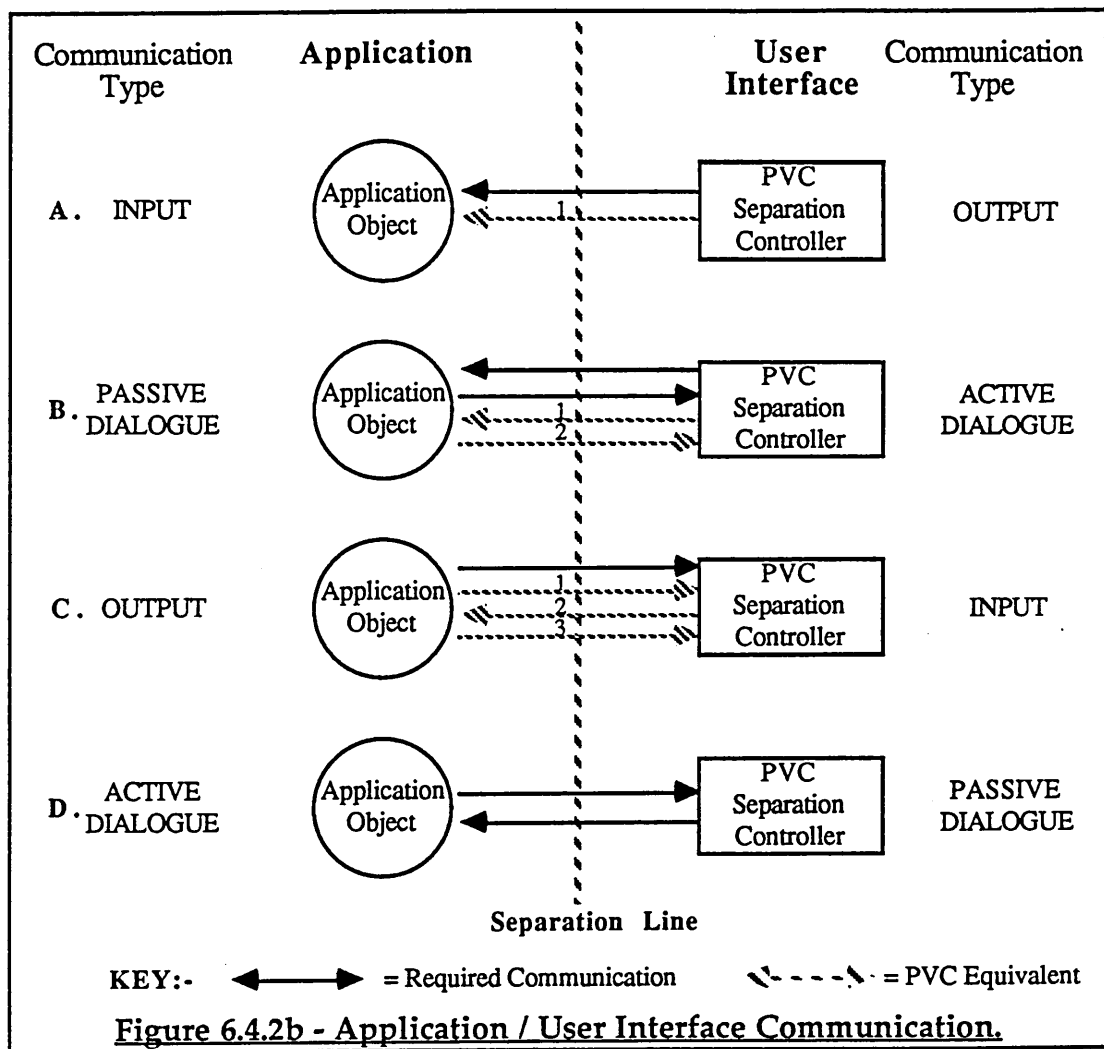


Figure 6.4.2b illustrates the four types of communication which may occur between application and interface. These types are named relative to both PVC and Application Object, and communication is shown using arrows. In each case, except type D, two types of arrow are used to distinguish between actual communication, and the PVC equivalent. Examining each type in detail :-

- A** This communication is initiated by the PVC, and is used to inform an Application Object of any state changes that result from user interaction. The communication is one way, and the Separation Controller does not expect any reply from the Application Object. This is achieved using an output Linkage Slot which contains a Smalltalk message to be sent to the Application Object, along with any necessary arguments. This message is sent via the Separation Controller as illustrated with arrow 1.

- B** This communication is again initiated by the PVC, and is used to request information from the Application Object. The PVC then has to wait for the object to return the appropriate item of information. This is achieved using an input Linkage Slot containing a Smalltalk message, which does not usually require arguments. The Separation Controller sends this message to the Application Object (arrow 1), and waits for the Application Object to reply with the relevant information (arrow 2). The returned information takes the form of a Smalltalk object.
- C** This type of communication is initiated by the Application Object. It occurs when the Application Object wishes to inform any active PVCs that a change has occurred which may affect the representation displayed by their View component. This is achieved using object dependency. Whenever a PVC is activated, it is made dependent upon the Application Object to which it is 'plugged'. This dependency does not affect the Application Object. Whenever an Application Object wishes to inform active PVCs of a state change, it must send the *changed* or *changed:* (with an argument) message to itself. This is then processed by the Smalltalk dependency mechanism as described in appendix B. The Separation Controller receives the *update* or *update:* (with the same argument) message accordingly, as shown by arrow 1. If the *update* message is received, then all input Linkage Slots are notified that their attached Application Object may have changed in a way that affects the value returned by them. If the *update:* message is received, then only the input Linkage Slots whose value matches the argument which is passed are notified of possible changes. The affected input Linkage Slots then request new information as shown by arrows 2 and 3. Finally, the PVC View updates itself accordingly.
- D** This type of communication is again initiated by the Application Object, which requests certain information to be input by the user. This was not implemented, and is further discussed in chapter eight.

Often, an Application Object changes only a small part of its state, and the message *changed:* is used to inform dependents. The Application Object also supplies the names of the methods which are affected by a particular state change. For each method, the message *changed:* with the method name as an argument, must be sent to itself. The wider the effects of the change to an object, the more likely it is that the *change* message will be most efficient.

This can be illustrated using the Person Class example from appendix D. The method which updates the age of a Person (*age:* or *dOB:*) must inform any dependents of the change in one of two ways :-

- (a) Two messages :- *self changed: age* and *self changed: dOB*.
- (b) One message :- *self changed*.

Changing the age of a Person affects the response by that Person to the *age* and *dOB* messages. Method (a) uses two *changed:* messages, one for each of the two effects. In each case the message affected determines the argument used. Method (b) is more general and uses only one message, which informs any dependents that the entire Person instance may have changed. Method (a) requires that only dependents which use the two messages *age* and *dOB* update themselves accordingly. However, method (b) requires that all dependents, regardless of which messages they use, update themselves.

This constrains the application designer who has to describe the effects of changes to an Application Object using the *change* and *changed:* messages. This may not be as limiting as first appears, because only the direct Application Object effects need be defined. The PVC mechanism also imposes the constraint that the object to which it is attached responds to all of its Linkage Slot message values.

The Separation Component provides specific methods which enable communication to occur between Application Object and PVC. For example, consider a PVC with an input Linkage Slot called <inputSlot>, an output Linkage Slot called <outputSlot>, and an I/O Linkage Slot called <inputOutputSlot>. Whenever input is required, the message *input:* should be sent by the View or Interaction Controller to its Separation Controller, with the Slot name as an argument. For example, *input: inputSlot* or *input: inputOutputSlot*. This then returns the appropriate item of information. If required the message *currentValue:* may be used to return the current or previous slot value. This method does not perform any actual model input,

and simple returns the current slot value, which was determined when the last *input:* message was issued. Whenever output is required the message *output:* should be used, e.g. *output: outputSlot* or *output: inputOutputSlot*. If the output message requires arguments, then these arguments can be sent by using the alternative method *output:using:*, with the appropriate arguments, e.g. *output: outputSlot using: 123*. The Separation Controller then handles the relevant Application Object communication according to the knowledge it contains. The implementation and use of Linkage Slots by a PVC is described in more detail in section 6.4.5.

6.4.3. Defining a Direct Manipulation Interface for an Object.

The architecture uses two types of interface components, which are based upon the PVC mechanism. A Part Pluggable View Controller (Part PVC) is used to describe the entire interface, while Interaction Pluggable View Controllers (Interaction PVC) are used to describe the individual elements from which it is comprised. Interaction PVCs represent specialised interface components, and examples include buttons, menus, graphs, sliders, and switches, while Part PVCs are analogous to a window which contains the Interaction PVCs. A Part PVC is attached to an Application Object, while Interaction PVCs are attached to a Part PVC. As discussed later, Application Objects may appear as aggregate parts of other Application Objects, hence the term Part Pluggable View Controllers. In such situations, a Part PVC may also be attached to another Part PVC in a similar way to Interaction PVCs. To summarise, the Part PVC effectively handles screen output, and user input for each Interaction PVC or Part PVC attached to it. It also controls the communication between different Interaction PVCs, and between the Application Object and the interface.

For every interface there is a corresponding Part PVC description which is attached to an Application Object using a Class instance method with a unique Part PVC title. This grammatical description defines the component Interaction PVCs, how they are related to one another, and specifies any communication with the Application Object. A user can use any of the Part PVC descriptions defined for an Application Object Class, to interface with it. When a Part PVC description is selected, it is executed by the UIMS and the appropriate Direct Manipulation interface generated. The Direct Manipulation interface can then be used by the user. A Part PVC description that has been executed and is being used is known as an active Part PVC.

Similarly, the Interaction PVCs which make up an active Part PVC are known as active Interaction PVCs.

6.4.4. Part Pluggable View Controllers.

Figure 6.4.4 illustrates the component structure of a Part PVC, which is based on the PVC mechanism. However, with regards to the actual implementation of a Part PVC, the View and Separation Controller are combined together into one component. Any communication between the Application Object and the Interaction Controller travels through the View, and effectively through the Separation Controller. This simplifies the Smalltalk Code required for implementation, but does not affect the functioning of the theoretical model.

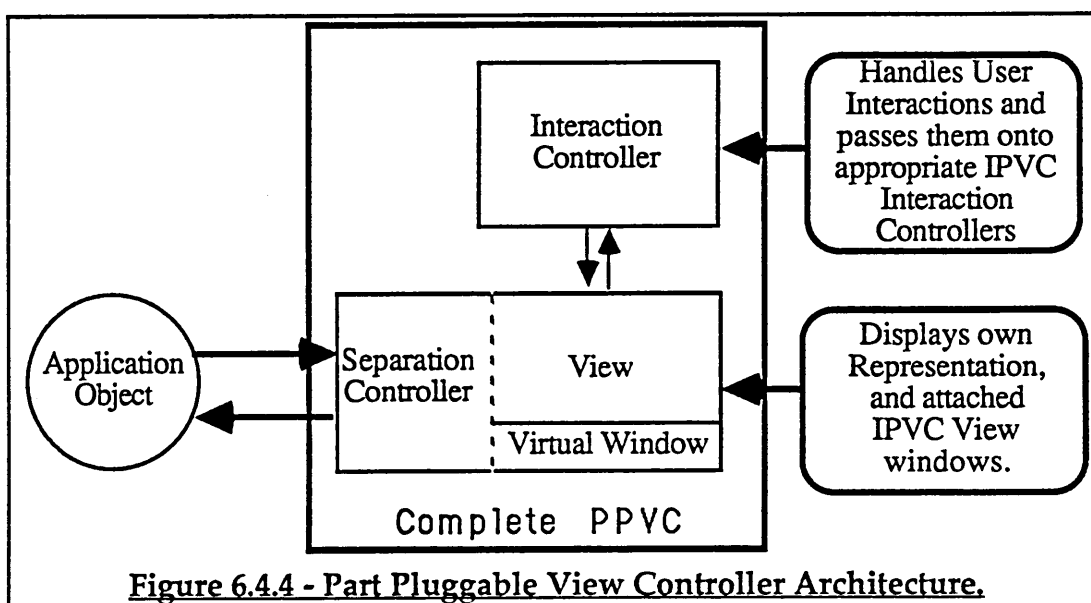


Figure 6.4.4 - Part Pluggable View Controller Architecture.

Direct Manipulation interfaces are built by combining Interaction PVCs, attaching them to a Part PVC, and placing them appropriately within the Part PVC virtual window. Effectively, the Part PVC serves as a container for Direct Manipulation interface Interaction PVCs, and as a Separation Controller. All Interaction PVCs attached to the same Part PVC share the same Application Object. Interaction PVCs and Part PVCs are defined using Classes, and re-use is easily achieved by creating new instances.

For every Application Object, the user is presented with a choice of interfaces which may be used, each of which correspond to a Part PVC description. Each description is implemented as a Smalltalk method which is attached to the Application Object Class. The method name identifies the Part PVC or interface name. This Part PVC method code has a strict format which is

described using Extended Backus Naur Form in appendix E. This code can be implemented directly by the interface designer, or can be automatically generated using the UIMS integrated interface design Tool-set described in section 6.5.

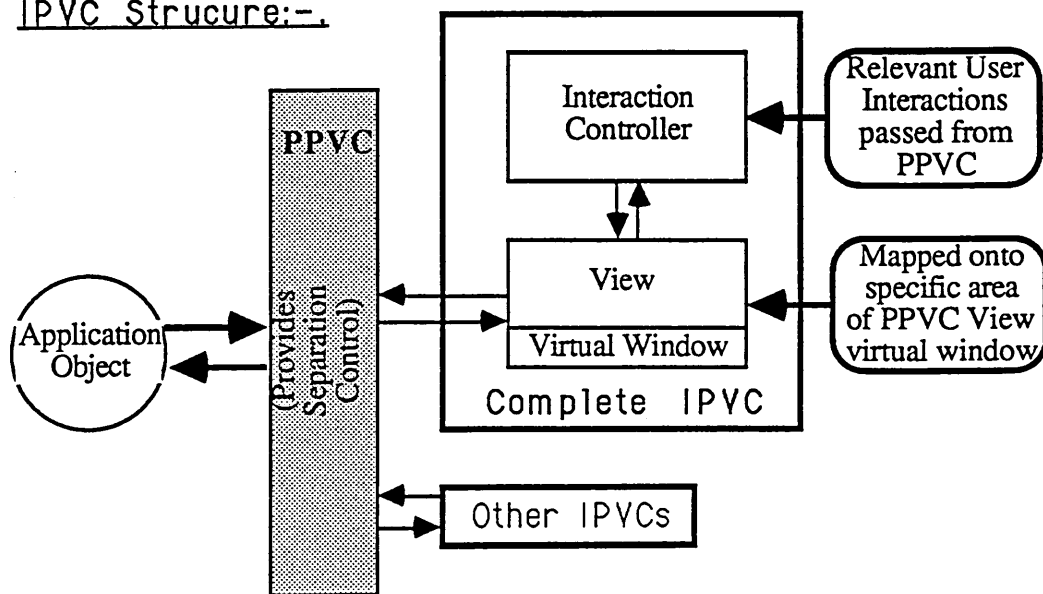
A Part PVC groups together Interaction PVCs and acts as their Separation Controller. A Part PVC View component virtual window is used to contain individual Interaction PVC View virtual windows, and provides suitable scaling, translation and display functions. Every Interaction PVC which is attached to a Part PVC defines the position of its View component virtual window, within the Part PVC View component virtual window. This assumes a Part PVC virtual window X Y coordinate system which is scaled in a range from 0 to 1. Whenever a Part PVC is used, it then automatically scales the virtual window of its View component to the specified physical screen area. User interaction is also monitored by a Part PVC, and passed on to the appropriate Interaction PVC Interaction Controller. This must then take the necessary actions.

Although basic abstract Part PVC View and Interaction Controller Classes are provided, special Part PVCs may implement additional functions. These special Part PVCs must be implemented as Sub-classes of the basic abstract Part PVC Classes, and must again use Linkage Slots in the usual way to provide any extra communication requirements with the Application Object. This facility is described in section 6.4.11.

6.4.5. Interaction Pluggable View Controllers.

As for Part PVCs, each Interaction PVC uses the View and Interaction Controller structure from figure 6.4.1. However, the Separation Controller function is now provided by the Part PVC to which an Interaction PVC is attached. Figure 6.4.5 illustrates the structure of an Interaction PVC, and shows how individual Interaction PVCs are combined in order to generate complex Direct Manipulation interfaces.

IPVC Structure:-



Connecting IPVCs to a PPVC:-

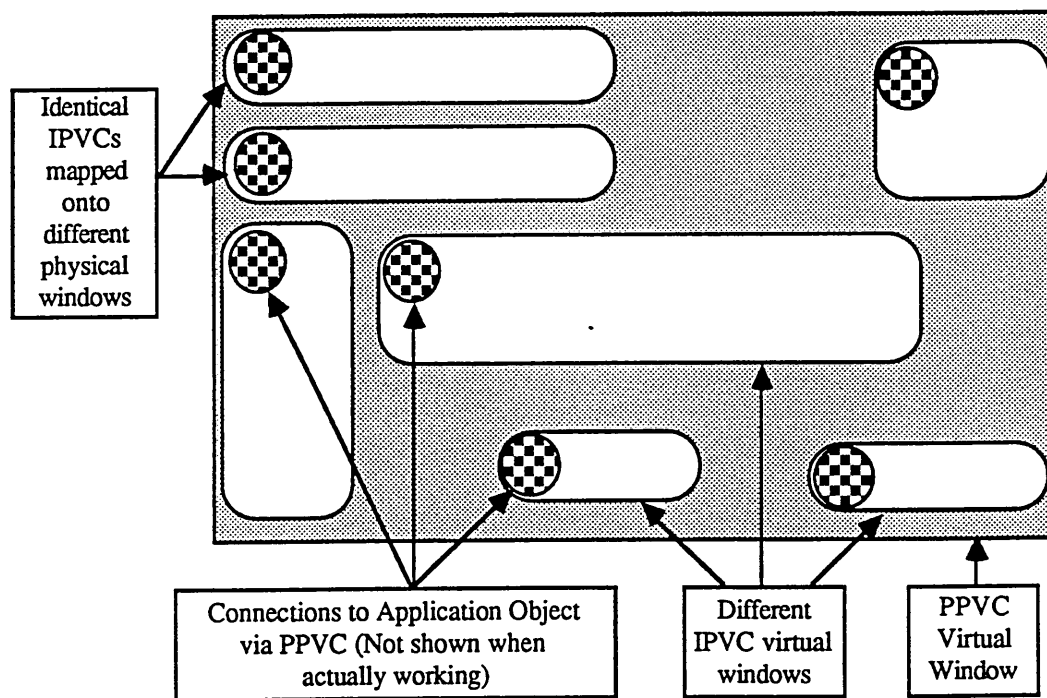


Figure 6.4.5 - Interaction Pluggable View Controllers Model, and connection to a Part Pluggable View Controllers.

Each Interaction PVC represents some type of interaction component, for example a button, switch, bar chart, or tree diagram. It contains a set of Linkage Slots which describe the communication required for it to work. These linkages are made to an Application Object through the Part PVC to which the Interaction PVC is attached. All Interaction PVCs attached to the same Part PVC effectively share the same Application Object. Each Linkage Slot is described using a special syntax which is detailed below in section 6.4.5.2.

An interface is implemented by selecting Interaction PVCs and attaching them to an Application Object Part PVC. This attachment, or 'plugging' is achieved by specifying the values contained in the Interaction PVC Linkage Slots, and finally by defining where the Interaction PVC appears in the Part PVC window, and its relative size.

Once attached to a Part PVC, the Separation Controller function for an Interaction PVC is provided by the Part PVC, and any communication between Application Object and Interaction PVC occurs through the Part PVC. When an active Part PVC is being used, the user is provided with a pointer which is moved around the screen using the mouse device. Whenever the pointer is over the physical window of an Interaction PVC (mapped onto the Interaction PVC virtual window), the Part PVC sends any user interactions to that particular Interaction PVC. The Part PVC uses a specific set of messages for informing of user interactions, and the Interaction PVC Interaction Controller must be implemented to respond to these messages appropriately.

A set of Interaction PVCs has been developed, and these are listed in appendix F. If Interaction PVCs are required which are not in this set, then the interface designer has simply to create new Interaction PVC Class descriptions using the outline described below in section 6.4.5.1. A library of generic Interaction PVCs can therefore be built up.

6.4.5.1. Internal Interaction Pluggable View Controller Structure.

The internal structure of an Interaction PVC is based on a specific framework. This framework will be of interest to the interface designer who needs to implement new Interaction PVCs. The framework is implemented using two Smalltalk abstract Classes. One Class implements the Interaction Controller structure, and the second the View. Although implemented separately, the View and Interaction Controller Class are linked together by information contained in the View. An Interaction PVC is used by generating a new instance of the View Class. This automatically creates the relevant Interaction PVC Interaction Controller Class instance which is linked to the View. Effectively, the same Interaction PVC Interaction Controller Class can be used by different Interaction PVC Views, and vice versa. The View and Interaction Controller Class instances may communicate with one another. This may be initiated by either, and suitable

Smalltalk methods are provided. The Linkage Slots which define Application Object communication are contained within the View Class. Any communication between Interaction Controller and Application Object must therefore pass through the View. Again, this consolidation does not affect the theoretical functioning of the PVC mechanism, and simplifies the code required.

A basic View and Interaction Controller abstract class have been implemented, and these provide default Interaction PVC methods. They also specify any methods which must be implemented in subsequent Interaction PVC Sub-classes. New Interaction PVCs must be implemented as Sub-classes of these Interaction Controller and View abstract Classes. They may also be implemented as Sub-classes of other Interaction PVC Interaction Controller and View Classes, which are themselves Sub-classes of the two abstract Classes. This utilises the benefits of inheritance, as new Interaction PVCs can re-use code from existing Interaction PVC Super-classes. New Interaction PVCs may also contain code which is copied from other Interaction PVCs.

The complete basic Interaction Controller and View Interaction PVC abstract Class implementations are too extensive to include as a code listing within the appendices. However, the complete Class hierarchy is given in appendix G, while the actual Smalltalk 80 Interaction and Part PVC Class implementations are available for inspection in listing, or code format, upon request from Sheffield City Polytechnic.

All communication between View and Application Object is controlled by the Part PVC to which it is attached. This takes the form of Linkage Slot input and output calls. The Part PVC directs any user interactions to the appropriate Interaction PVC Interaction Controller, which must then interpret them. The Interaction Controller message protocol is therefore fixed and the method code should be re-defined accordingly.

The View essentially provides three message protocol sub-sets. One deals with the Linkage Slots, a second with the handling of Application Object state changes, and a third with displaying information on the screen. The Linkage Slot sub-set can be used for each Linkage Slot, and the relevant Linkage Slot name must be specified. The Linkage Slot sub-set method code should not be re-defined in any of the Sub-classes.

Application Object state changes are handled automatically by the Part PVC, which expects a specific Interaction PVC View message protocol. An Interaction PVC View should provide a message called *changedSlotName* for every input or I/O Linkage Slot, where *SlotName* is the name of the Linkage Slot, e.g. *changedcurrentItem*, or *changedcurrentList* for Linkage Slots *currentItem* and *currentList* respectively. The appropriate method is then called when the attached Application Object changes in such a way that may have affected a specific input Linkage Slot. As the object change may in fact have not affected the value returned by a particular input Linkage Slot, the following method code is recommended. This first checks whether the linkage slot value has actually changed, before performing appropriate update functions :-

changedSlotName

```
(self currentValue: #SlotName) = (self input: #SlotName)
ifFalse: [UPDATE FUNCTIONS]
```

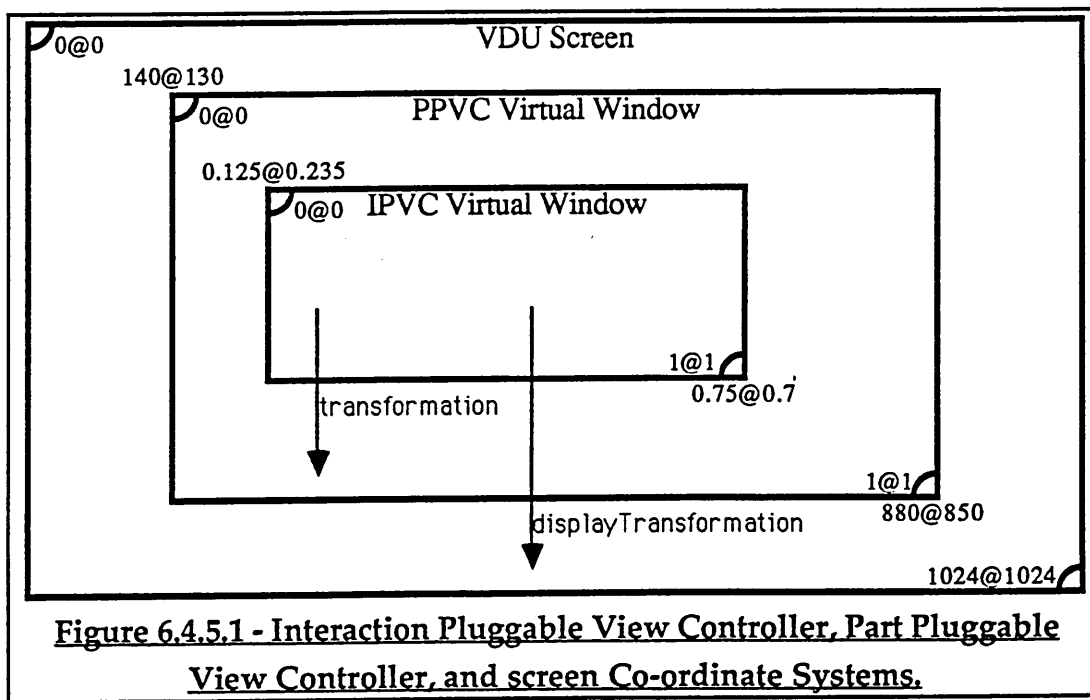
Whenever an Interaction PVC is used for the first time, or the entire Application Object changes (i.e. the Application Object issues the *changed* message) several message are sent to the Interaction PVC in a strict order, these are *modelInitialize: oldObject*, *initializeInputValues*, and *displayAll* (where *oldObject* is the previous Application Object before its change, or nil when the Interaction PVC is being used for the first time). The corresponding method code for these messages should be re-defined in the Interaction PVC View Sub-classes. The default code for *modelInitialize:* and *initializeInputValues* is empty, i.e. does nothing.

Every View provides a rectangular area to which text and graphics can be written. This area is mapped onto the screen window by the Part PVC, when the Interaction PVC is activated. The Interaction PVC designer need not be concerned with the actual physical screen window as the mapping from virtual to physical window is handled automatically. The coordinates of this virtual window may be defined in several coordinate systems :-

- in its own coordinate system
- in the Part PVC coordinate system
- in the coordinate system of the screen.

Each View has an internal coordinate system which can be mapped onto the coordinate system of the Part PVC View to which it is attached, and onto the coordinate system of the screen itself. This is illustrated in figure 6.4.5.1, where the coordinates belonging to a View are indicated inside the View, and the coordinates of that view within its Superview are shown outside. In order to move from one view coordinate system to another, simple geometric transformations are implemented. The Smalltalk *WindowingTransformation* Class serves to represent these geometric transformations. Each View has two instances of this Class, to transform between the coordinate system of the View and both the Part PVC and physical screen coordinate systems.

A specific method protocol is expected when generating the display for a View, and this should be followed when implementing new Interaction PVCs. One method called *modelDisplay* is expected, and is called whenever the Interaction PVC is first displayed or when the attached Application Object changes in a major way as indicated by the use of the general *changed* rather than the more specific *changed:* message. Other methods should also be implemented which are called whenever an Application Object changes its state in a way which affects only part of the view generated by an Interaction PVC View. These can then be called by the *changedSlotName:* method code, and may also be used by *modelDisplay* method to display the initial representation.



Finally, two other View instance methods are expected, *initialize* and *release*. The *initialize* method is called once whenever a new Interaction PVC is first added to a Part PVC, and *release* is called when an Interaction PVC is closed. Default methods are implemented within the basic View abstract Class, and any Sub-classes which override these methods must ensure that they first call the abstract View Class methods, using the *super initialize*, and *super release* messages.

6.4.5.2. External Interaction Pluggable View Controller Description.

The Linkage Slot requirements for each Interaction PVC are specified using an external description defined by the View Class. This is used by the interface design Tool-set when adding new, or modifying existing Interaction PVCs. Every View Class definition must respond to the Class message *SlotDescription* with the Interaction PVC external description. The response to *SlotDescription* should be an array which describes the format of individual Interaction PVC Linkage Slots. The syntax for this description is now defined and explained using examples from appendix F. Appendix E also gives the full Extended Backus Naur Form syntax.

Consider the List and BarChart Interaction PVC examples, and figure 6.4.5.2

:-

List IPVC

Linkage Slot Descriptors *SlotDescription*: :-

```
((currentItem 'Current Selected Item' (Input Message with (0)
noMsgArgs))
 (newSelection 'Informing Model of new selection' (Output
Message with (1) noMsgArgs))
 (itemList 'Item List' (Input Any Collection with (0)
noMsgArgs) '#()')
 (stringPrint 'Print List as String ?' (Input Literal Boolean))
 (oneItem 'Only One Item in List ?' (Input Literal Boolean))
 (yBM 'Interaction Menu' (Input Any Menu with (0)
noMsgArgs)))
```

Bar Chart IPVC

Linkage Slot Descriptors *SlotDescription* :-

```
((label 'Title of Chart' (Input Any String with (0) noMsgArgs)
  ""No Label")
 (barValues 'Bar Values' (Input Any Number with (0)
  noMsgArgs) multiSlot)
 (barTitles 'Bar Labels' (Input Literal String) ""No Label"
  multiSlot)
 (yBM 'Interaction Menu' (Input Any Menu with (0)
  noMsgArgs)))
```

Input Linkage Slots may be typed. This type identifies the Class type of the Smalltalk object which the Linkage Slot message returns to the Interaction PVC. It is used interactively to request a new, or modify an existing Interaction PVC Linkage Slot literal value.

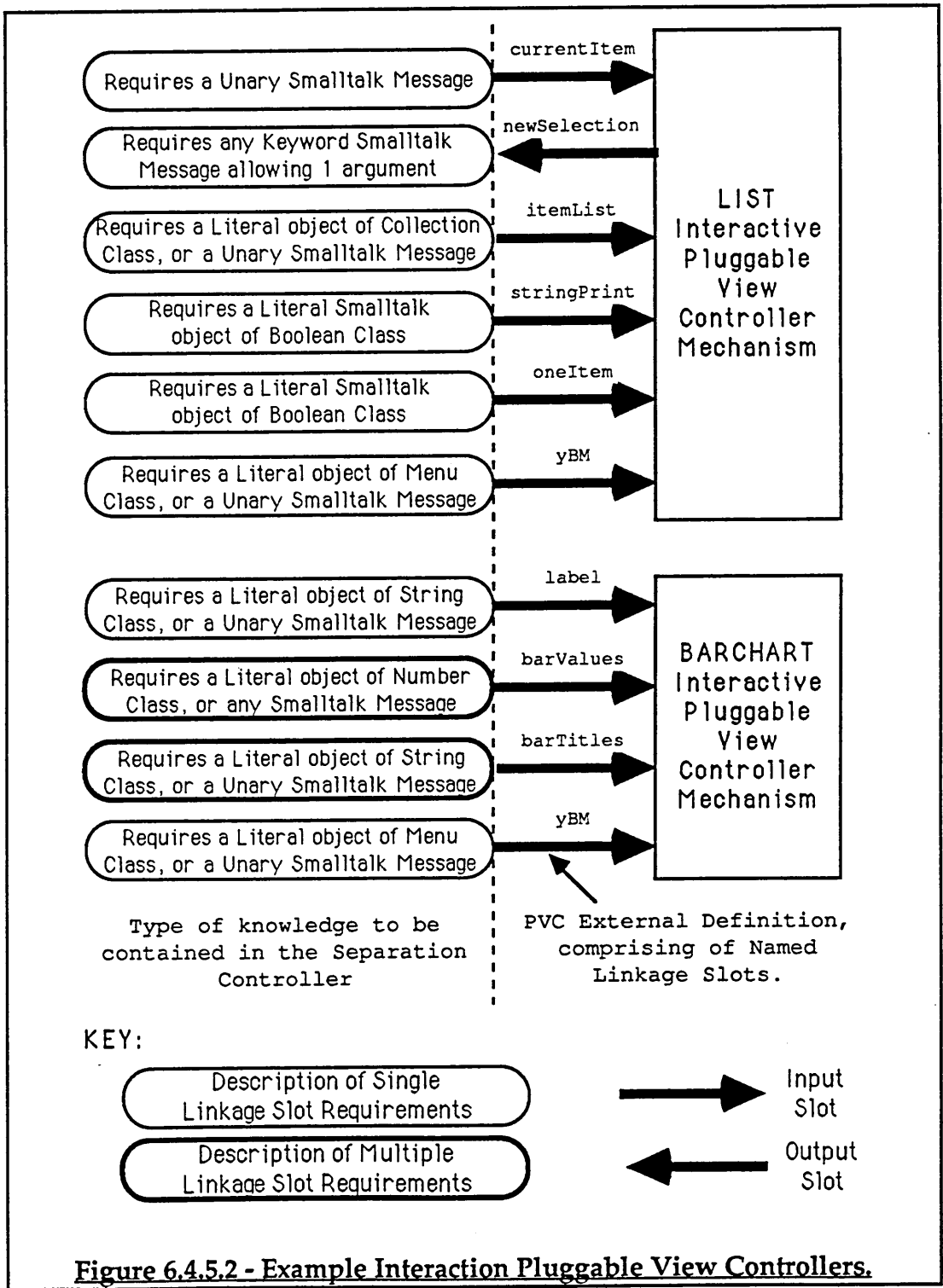
The Interaction PVC designer must describe all Linkage Slots, and implement the Class method accordingly.

Every Linkage Slot descriptor comprises an array containing 3 conditional items, and two optional items in a specific order. The three conditional items are Slot Name, Slot Title, and Slot Type in that order. The two optional items which follow the conditional ones are Slot Default Value, and Multiple Slot ID.

The Slot Name identifies the Linkage Slot name (e.g. <currentItem> and <barValues>), while the Title represents a description of the Linkage Slot which takes the form of a string (e.g. <'Current Selected Item'> and <'Bar Values'>). This Title is used by the Tool-set when adding an Interaction PVC to a Part PVC.

The Slot Type consists of an array which identifies what type of communication the Linkage Slot is used for, and what type of value it can contain. The first item in the array identifies the Communication Type and must be either <Input>, <Output> or <IO>, implying either input, output, or input and output in relation to the Interaction PVC. The next item is known as the Value Type, and identifies the type of value the Linkage Slot must contain. The Value Type can be either <Literal>, <Message>, or <Any>.

However, when the Communication Type is defined as <Output>, the Value Type must be set to <Message>, as Literal values are not allowed. A Value Type of <Literal> implies that the Linkage Slot must contain only Literal values, and a third and final item in the array must identify what type of Literal value may be contained, e.g. <(Input Literal Boolean)>. This must correspond to an existing Smalltalk Class, and is used by the Tool-set to acquire a Literal value when the Interaction PVC is being added to an existing Part PVC.



A Value Type of <Message> signifies that the Linkage Slot must contain a Smalltalk message. As a result, three further array items must be included to identify the type of Smalltalk message allowed, e.g. <(Input Message without () noMsgArgs)>. The first of these may be either <with> or <without>, and the second must be an array of positive integers, which may be empty. These two items are combined to decide what type of Smalltalk message the Linkage Slot can contain. Smalltalk messages may or may not require arguments. The numeric array represents a list of argument counts for Smalltalk messages, where 0 implies no arguments. If <with> is specified, then this array represents which Smalltalk messages are allowed, e.g. <with (0 1 2 3)> implies that only messages which require 0, 1, 2, or 3 arguments are allowed. If <without> is used, then the array represents the Smalltalk messages which are not allowed, e.g. <without (0 1 2 3)> implies that any Smalltalk messages are allowed, except those which require 0, 1, 2, or 3 arguments. It follows that <with ()> is an invalid entry, implying that no Smalltalk messages are allowed. This should not be used.

The final additional item determines whether the Linkage Slot message arguments (if any) are to be included in the Part PVC description, and therefore requested when adding new or changing existing Linkage Slot values. Its value may be either <noMsgArgs> <msgArgs> or <nilMsgArgs>. The value <noMsgArgs> implies that if arguments are required, they are not requested when the Interaction PVC is added. This implies that the Interaction PVC will determine its own arguments. The value <msgArgs> implies that the arguments required by the selected Linkage Slot message are to be fixed, and specified within the Part PVC description. The arguments must therefore be requested when the Linkage Slot value is added or modified. They are requested in matching key-word order using the 'Please type argument n' prompt. For each argument, the interface designer is presented with an editor window in which to type Smalltalk code. This code is executed, and the value returned as the appropriate argument. For example, consider the Smalltalk message *abcd:efgh:ijkl:mnop:* which requires 4 arguments. The user would be prompted for these 4 arguments, and the order would match the message order, i.e. *abcd: argument1 efgh: argument2 ijkl: argument3 mnop: argument4*. Finally, <nilMsgArgs> implies that arguments are specified in the Part PVC description, but they are all set to a value of nil.

The final Value Type option <Any> implies that a Linkage Slot can contain either a Literal value, or a Smalltalk message. Four additional array items are then required, e.g. <(Input Any Collection with (0) noMsgArgs) >. The first identifies the type for any Literal value, while the remaining three identify any Smalltalk message type in an identical way to the three additional items included when a Value Type of <Message> is used.

The optional Default Value determines what default value is returned by an input Linkage Slot. This default value is used whenever the returned value is nil. This occurs when either the Linkage Slot value is nil, or the Application Object returns nil in response to a PVC input request. It takes the form of a piece of Smalltalk code enclosed by single quotes which is executed to determine the default value, e.g. '12', 'Array new with: 3 with: 5 with: 6', '1 to: 30 by: 5'. If no default is stated, then the returned nil value is used.

Finally the Multiple Slot ID may be included and takes the form of the verb <MultiSlot>. This implies that the Linkage Slot may contain Multiple Values, and is discussed in 6.4.8 below. If this optional verb is excluded then the normal Single Linkage Slot is assumed.

6.4.6. Communication Between Interaction Pluggable View Controllers.

Interaction PVCs connected to the same Part PVC may communicate directly with one another. When the interface is defined, the designer has the option of linking different Interaction PVCs to one another. An Interaction PVC may be connected to one or many Interaction PVCs. Whenever a link is made between Interaction PVC A and Interaction PVC B, any links that already exist between A and other Interaction PVCs are passed onto B. Similarly, any of existing links belonging to B are passed onto A. This type of relationship is similar to the Smalltalk object dependency mechanism, but may only be used for Interaction PVCs.

Messages are provided by the Interaction PVC abstract Classes, to enable these Interaction PVC links to be utilised. How these messages are used, and the effect of linking different Interaction PVCs together is decided by individual Interaction PVC implementations. For example, a bank of switches may be created by linking many individual Interaction PVC switches together. The effect is similar to a set of TV station selector switches, where only one can be selected at a time, and selecting one de-

selects the currently selected switch. These inter Interaction PVC link messages can be instantiated by an Interaction PVC View sending the relevant message to itself :-

allLinks

This returns an array of pointers, pointing to the linked Interaction PVCs including the Interaction PVC which sent the message. This array of pointers can then be used to communicate with the linked Interaction PVCs.

myLinks

This returns an array of pointers, pointing to the linked Interaction PVCs excluding the Interaction PVC which sent the message. This array of pointers can again be used to communicate with the linked Interaction PVCs.

isLinked

This returns true, or false depending upon whether an Interaction PVC is linked to any other Interaction PVCs.

Each Interaction PVC determines whether it can be linked by its response to the View Class message *IsLinkable*. This must return one of three verbs, <None>, <Identical>, or <Any>. The verb <None> implies that an Interaction PVC cannot be linked to any other Interaction PVC. This is the default response provided by the example Interaction PVC View Class. The verb <Identical> implies that an Interaction PVC View can be linked to any other Interaction PVC View of the same Class as itself. Finally the verb <Any> implies that an Interaction PVC View can be linked to any other Interaction PVC View attached to the same Part PVC, regardless of its Class.

An Interaction PVC links itself to another Interaction PVC by sending the appropriate message to the other Interaction PVC, and setting the argument (*anIPVC*) to itself. A differentiation is made between the linking together of Interaction PVCs interactively at run time (i.e. when an interface, or Part PVC is being interactively designed), and when a Part PVC description is being executed (i.e. after the link has been interactively defined and the Part PVC description automatically created). The abstract Interaction PVC View Class implements the following messages which may be re-defined in subsequent Interaction PVC View Sub-classes.

addLinkInteractive: anIPVC

This message links anIPVC to the message receiver, i.e. another Interaction PVC. It is used to link together two Interaction PVCs at run time. That is once a Part PVC has been executed, and is being interactively modified.

addLinkCreation: anIPVC

This message again links anIPVC to the message receiver. It is used to link together two Interaction PVCs at execution time. That is while a Part PVC description is being executed in order to generate a Direct Manipulation interface. Once an Interaction PVC link has been defined interactively it is stored as part of the Part PVC description, and this message is used to recreate the linkage when the Part PVC description is re-used.

The reason for this distinction is due to state conflict resolution. When two Interaction PVCs are linked to one another, their different states may conflict, this needs resolving. For example, linking a switch which is on to a bank of other switches; only one banked switch is allowed to be on, and therefore either the switch being linked, or the conflicting banked switch which is on, must be switched off. While an interface is being used this conflict can be resolved by interrogating the user or interface designer. However, this type of interrogation is impractical during Part PVC execution because an interface may be being used by somebody who knows nothing about the design decisions which were made.

6.4.7. Interaction Pluggable View Controller Cursors.

This feature allows individual Interaction PVCs to define the cursor shown when the mouse pointer appears over their Interaction PVC window (after it is mapped onto the physical screen).

Smalltalk provides a special Class called *Cursor*, which defines graphic shapes which can be displayed on the screen. These shapes are used to represent the position of the mouse pointer, and move according to the mouse movement from the user. Individual Interaction PVC Views may implement the Class method *Cursor* such that it returns an instance of the *Cursor* Class, which defines their cursor. This cursor is then displayed whenever the mouse pointer is over their Interaction PVC View window.

As the mouse pointer moves out of the window, the previous cursor is restored. The default cursor is defined as an arrow.

6.4.8. Interaction Pluggable View Controller Multiple Linkage Slots.

Because the number of Linkage Slots is fixed by the Interaction PVC implementor, this feature can be used to provide dynamic Multiple Linkage Slot definition. This enables a Linkage Slot to contain more than one value, and the number of values may be altered at run time. A Multiple Linkage Slot uses a variable length array to contain the Linkage Slot values, i.e. either a Smalltalk message, literal, or nil. An example which uses this feature is the BarChart Interaction PVC, where the number of bars displayed is variable. It is not feasible to represent each bar value with a separate Linkage Slot. Instead, the values represented by each bar can be determined using a single input Multiple Linkage Slot. Multiple Linkage Slots may also be used for output, and input/output.

As described in section 6.4.5.2, an Interaction PVC Multiple Linkage Slot is described in a similar way to a normal Linkage Slot. However, the external Linkage Slot descriptor must include the verb <MultiSlot>. The external Linkage Slot descriptor applies to all multiple values, i.e. all values are of the same Communication Type (input, output, or input/output), same Value Type (Literal, Message, or Any), and share the same Default Value. A set of specialized messages are provided to enable the Interaction PVC implementor to determine how a Multiple Linkage Slot is used. These are in addition to the message interface described in section 6.4.5.2, and are implemented in the Interaction PVC View abstract Class *MultiIPVCView*. These messages are now listed. In each case *slotName* identifies the Interaction PVC Multiple Linkage Slot being modified, *value* represents the new, or modified value at a particular position (i.e. a Literal value, Smalltalk message, or nil), while all positions must be integers. Suitable error checking is also incorporated.

addMultiSlot: slotName value: newValue type: aType

This allows a new Linkage Slot value *newValue* of Type *aType* (either #Literal or #Message), to be added to the end of the existing Multiple Linkage Slot called *slotName*.

changeMultiSlot: slotName atPos: pos value: newValuetype: aType

This allows the Multiple Linkage Slot value for an existing position *pos*, to be set to *newValue*.

removeMultiSlot: slotName at: pos

This allows the *slotName* Multiple Linkage Slot value at position *pos* to be removed.

swapMultiSlot: slotName pos1: pos1 pos2: pos2

This enables the Multiple Linkage Slot values at *pos1* and *pos2* to be swapped.

multiSlotCount: slotName

This returns a count of the current number of values contained within a Multiple Linkage Slot.

input: slotName at: pos

This performs input, and returns the value for the particular Multiple Linkage Slot value at position *pos*.

inputAll

This performs input, and returns the value for all of the Multiple Linkage Slot entries, An Array of values is returned, which is ordered according the Linkage Slot order.

currentValue: slotName at: pos

This returns the current / previous value for the particular Multiple Linkage Slot value at position *pos*. No actual input is performed.

output: slotName at: pos

This outputs to the model using the current Multiple Linkage Slot value and arguments at position *pos*.

output: slotName at: pos using: object

This outputs the object *object* using the Multiple Linkage Slot value at position *pos*.

The current values for a Multiple Linkage Slot are stored when the underlying interface syntax description is automatically generated. These are then restored when the interface description is next used, or executed. When an Interaction PVC is added to a Part PVC, the first Multiple Linkage Slot value must also be specified. Once attached to a Part PVC which is active, further values for a Multiple Linkage Slot may be added. In the Bar Chart example, this would be the equivalent to adding more bars. The order of Linkage Slot values is maintained, with the last value added placed at the end of the array. Interactive tools are provided for removing, changing, and re-ordering the list of values contained in a Multiple Linkage Slot.

All Interaction PVCs which require Multiple Linkage Slots must be implemented as Sub-classes of the *MultiIPVCView* abstract Class. In doing so, several methods must be implemented to handle Application Object state changes which affect the Interaction PVC. These methods are similar to those described earlier in section 6.4.5.1 for single Linkage Slot, and are invoked by the Separation Controller whenever the Application Object state changes in a way which affects individual Multiple Linkage Slot values. Two methods are required, namely *changedSlotName*, and *changedSlotName: aPos*, where *SlotName* should be replaced with the name of the Multiple Linkage Slot, and *aPos* represents the number of the Multiple Linkage Slot value which is affected. The former method is invoked when the entire set of Multiple Linkage Slot values have been affected or changed, while the latter is invoked whenever the value returned by individual Slots may have changed. Examples include *changedBarTitles*, and *changedBarTitles: aPos*, for a Multiple Linkage Slot called *barTitles*.

6.4.9. Default Part Pluggable View Controllers.

Each Class defines the Part PVCs which can be used to interact with any instances of itself. Similarly, each Smalltalk Class may also define a default Part PVC to be used to interact with its instances. This takes the form of a Part PVC name, and may be changed using the Tool-set described later. This default Part PVC user interface is used in various ways throughout the system.

Every Smalltalk method returns an object upon completion of its code. The object returned may be specified using the Upward Pointing Arrow (^) character, and defaults to the receiver object. The object which is returned can itself be used as a receiver for other Smalltalk messages. For example `1 + 2 + 4` which returns 7. This is accomplished by sending the message `+` with the argument 6 (a Smalltalk object identified with the name 6) to the object titled 1. The object titled 6 is obtained from the result of sending the message `+` with the argument 4 to the object titled 2. All these objects are instances of the Integer Class. This concatenation of messages is one of the major strengths of Smalltalk. Unfortunately, if used incorrectly it also causes major problems. Objects are referred to by a name which points to them in memory. Object messages may return similar pointers to objects. In actual fact, an object may return a pointer pointing to one of its internal data fields. Once returned, this internal data field pointer (or rather the object it points to) may be the receiver of other Smalltalk messages. These chained messages may then alter the state of the internal data field without going through the message interface of the owner object. The internal data fields of a Smalltalk object can therefore be changed without the knowledge of the object itself. This breaks the encapsulation concept provided by object oriented languages.

Using an example from appendix D, consider the behaviour of an instance of the Person Class called `<freddy>`. The message *department*, when sent to `<freddy>`, returns an instance of the Department Class, which represents the department to which `<freddy>` belongs. The message *location:* with the 'new location' string as an argument may then be sent to this returned instance of a Department Class. Note that this Department Class instance has no named pointer, and the message *location:* is sent :-

freddy department location: 'the new department name'

This has allowed changes to be made to the internal data field called *department*, which belongs to a Person Class instance called `<freddy>`, and was performed without the knowledge of the owner to which the internal data field belonged, i.e. `<freddy>`. The correct way to access and change the location of the object pointed to by the *department* internal data field, would be to implement an instance method called *newLocation:* (or an equivalent name) in the Person Class. This method would take the new location as an

argument, and would then update the location for the department internal data field. It could also handle any side effects which may result.

Instead of returning pointers to internal data fields, Smalltalk should return pointers to copies of these fields. Although identical clones, these new objects would be separate from the actual internal data fields, and therefore not subject to this error. However, this inconsistency is often used to make Smalltalk code more efficient, and perform special types of functions. One way around this problem is to provide Part Hierarchies

Smalltalk does not usually provide the facility to build objects out of parts, i.e. Part Hierarchies. In this situation, the parts are themselves separate objects which are defined using Smalltalk Classes. As such they may be 'stand alone' instances, or attached as a part of an owner object. When attached as a part, their behaviour and state may change. Likewise, the state and behaviour of the owner may also change. One such Part Hierarchy implementation is discussed in chapter seven.

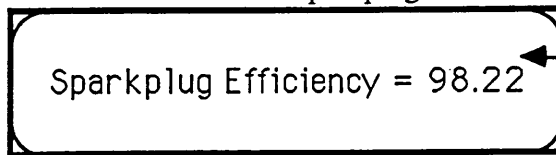
When an object Part Hierarchy exists, a Direct Manipulation interface may be required to provide access to the parts. Consider some of the examples from appendix D. The Sparkplug, Engine and Car Classes are implemented separately, and each Class may define its own set of Part PVC Direct Manipulation interfaces which can be used to interact with their instances. The Part PVCs defined for a part can be used when examining the part through a Part PVC defined for the owner. For example, we may define a Part PVC on a Car which allows the user to interact with the engine part belonging to a Car. Any of the Part PVCs defined on the Engine Class may be used in the Direct Manipulation interface defined to interact with the Car. The Engine Part PVC acts as a special type of Interaction PVC, and its Separation Control is provided by the Part PVC to which it is attached. This also handles the display and user interaction functions in a similar way to normal Interaction PVCs.

The effect of modifying a part, and how these effects are passed on to the owner are discussed in chapter seven. Needless to say, any effects that result from a change in an owner or part state are immediately seen in the affected PVC mechanism. Using figure 6.4.10, consider a Part PVC defined on a Car (C), which shows the acceleration, weight, and tyre pressures of the Car at a particular engine rev speed. The engine part is also shown using a Part PVC defined on the Engine Class (B). This Part PVC shows the engine rev speed,

and power being generated. It also provides an interface to the spark plug part using a Part PVC defined on the Sparkplug Class (A). This final Part PVC shows the efficiency of a sparkplug, and allows it to be modified. Any changes to the efficiency of a sparkplug immediately affects the power of the engine, and the acceleration of the car. These effects will be shown immediately in the relevant PVC Views. If an instance of the same Sparkplug Class, which was not part of an engine but a 'stand alone' object, was being interfaced with, the same Sparkplug Part PVC interface could be used. In this case, no effects would be passed on to a part owner object.

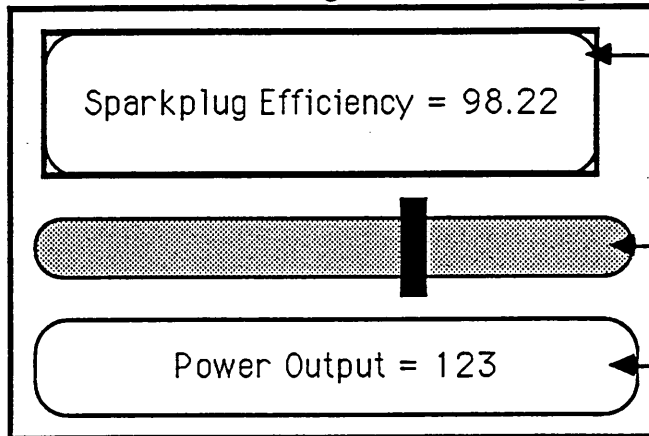
Using Part Hierarchies, Interaction PVCs are allowed access to part behaviour at lower levels. An Interaction PVC attached to a Part PVC may access the application messages belonging to the Application Object being accessed through the Part PVC. For example, a Bar Chart Interaction PVC may be attached to a Part PVC on a Car. This Bar Chart may then access individual wheel parts, and enquire on their pressure. The results may then be displayed as four bars in the Bar Chart. The Bar Chart Interaction PVC is attached to the Part PVC which is defined on the Car, and the Part Hierarchy allows 'safe' access to the wheel parts without breaking any data hiding or encapsulation rules. If this were not allowed, then the Car Class would need to implement some of the wheel part behaviour within its own behaviour, which would defeat the concept of Part Hierarchies.

A: PPVC defined on Sparkplug Class:



General I/O IPVC showing Sparkplug Efficiency, and allowing it to be modified

B: PPVC defined on Engine Class (including above Sparkplug Class PPVC):

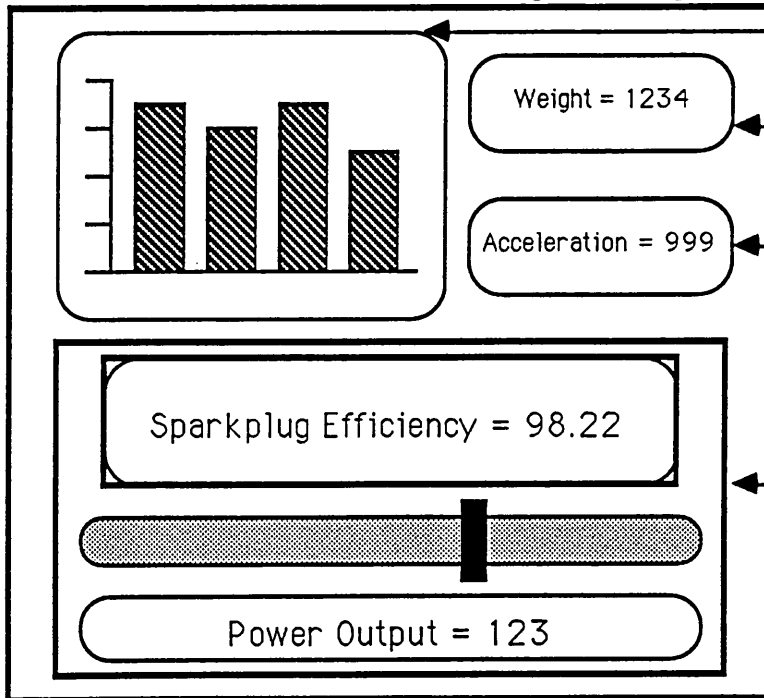


Sparkplug PPVC used to view sparkplug Part

Slider IPVC used to display and vary engine revs

General Output IPVC showing Power Output

C: PPVC defined on Car Class (including above Engine Class PPVC):



Bar Chart IPVC on tyre pressures

General Output IPVC showing Car weight

General Output IPVC showing acceleration

Engine PPVC used to view engine Part

Figure 6.4.10 - Combining Part Pluggable View Controllers.

6.4.11. Special Part Pluggable View Controllers.

PVCs have their own functionality which can be distinguished from that of their attached Application Object. This can serve as a separate Non-Interactive Function Core, on to which further Interaction PVC mechanisms can be linked.

For example, consider an Interaction PVC which displays a graphical representation of a tree. Assuming that only part of the tree is displayed on the screen, the Interaction PVC may provide functions for actions such as finding nodes and branches, and zooming in and out. The Interaction PVC may itself have further Interaction PVC components attached to it, for invoking these functions. For example, there may be buttons to zoom in and out, a slider to represent and change the percentage zoom, and a text editor to search for named nodes and branches. These further Interaction PVC components would be linked to the tree Interaction PVC, and access its Non-Interactive methods.

The concept of Special Part PVCs is included to investigate the possibility of these type of PVCs. This step is completed, and has proven that once Special Part PVC concepts are extended to Interaction PVCs this facility can be provided for all PVCs.

A Special Part PVC is essentially a Part PVC which provides additional application functions which are not supported by the Application Objects to which they are attached. Consider the Array Class in Smalltalk, which represents an array of heterogeneous objects. This Class does not provide a pointer to point to the current array item being examined. However, if this feature is required by an interface the Part PVC must provide it. A Special Part PVC may implement extra functions which use existing Application Object functions to provide more complex application behaviour. Any extra Application Object communication must be provided by additional Part PVC Linkage Slots.

These extra Part PVC functions appear to the designer as part of the functionality of the Application Object. As such, Interaction PVCs attached to Special Part PVCs can use these extra functions. This facility is useful but an important conceptual problem arises as to whether an interface should be allowed to implement additional Application Object functions. Chapter

eight includes further discussion, as well as justification for moving this implementation into the Separation Controller.

Special Part PVC Views must be implemented as Sub-classes of the *PPVCSpecialView* abstract Class, while the Part PVC Interaction Controller remains the same as the normal Part PVC Interaction Controller. The new Special Part PVC View must also provide an instance method called *localMenuItems* which returns an array containing a list of the additional methods. This distinguishes between the functionality of the Application Object, and the additional Special Part PVC functions. Finally, they must respond to the Class message *SlotDescription* with their Linkage Slot description, as previously discussed.

An example Special Part View Class called *PPVCUserModelView* has been implemented to demonstrate this facility. This Class enables individual User Model methods and Classes to be interactively selected for modification - a feature not provided by the User Model implementation itself. The External Linkage Slot Description for this Class is given in appendix H.

6.4.12. Interactive Creation of Smalltalk Objects.

New instances of Smalltalk objects are normally created by sending the *new* message to a specific Class. However, this is insufficient when a Literal Linkage Slot value is required while adding a new Interaction PVC to a Part PVC. An interactive means of creating new instances is required. A universal Class method called *interactiveCreate: aString on: oldValue* was added, which can be understood by all Smalltalk objects. The *aString* argument determines the title which is used when requesting a new instance, while *oldValue* represents an existing instance of the Class which may be nil - implying that a brand new instance is required. The response to this message is expected to be a new instance of the Class, which has been interactively created. When new Classes are added, the programmer should re-define this method appropriately. The default response is intended to present an editor window into which Smalltalk code can be typed. The result of executing this code is then returned as the new Smalltalk object. The method was implemented for other Classes such as Boolean which responds by asking whether true or false is required. Similarly, the String Class implementation is different. This displays the existing value *oldValue*, or ' ' if it is nil, in an editor window. The user is then allowed to modify the

window contents, and the end result is returned as a new String with the appropriate value.

This method is used when an Interaction PVC Literal Linkage Slot value is required, or is being modified. Finally, the method code can itself use Part PVCs defined on the Class.

6.4.13. Construction and Interaction Menus.

Smalltalk 80 provides three mouse buttons namely red, blue and yellow, selected by combining keyboard and mouse button presses. All PVCs have the option of providing a construction and interaction menu associated with the blue and yellow mouse buttons respectively. The construction menu is for use by the interface designer when building Direct Manipulation interfaces. Some example entries would be 'modify size', 'add links', and 'close Interaction PVC'. The interaction menu is for the user when performing normal interactions with the interface. Example entries for a List Interaction PVC include 'Add list item', 'Remove List Item', and 'Re-order List'.

Both menus must use the Extended Lean Cuisine Syntax described below, and may be generated using the design Tool-set. The functions associated with individual options must map onto the functions provided by the relevant PVC View Class, and may also be inherited from any Sub-class hierarchy.

The abstract Interaction PVC View Classes described earlier implement default interaction and construction menus for all Sub-classes. The options provided by these menus are listed in appendix I.

6.4.14. Extended Lean Cuisine Hierarchic Menus.

All UIMS menus are defined using an Extended Lean Cuisine (ELC) Hierarchical Menu syntax. This is based upon an extension of the 'Lean Cuisine' formal menu syntax described by Apperley [Apperley, M.D:1989]. Appendix E gives the full Extended Lean Cuisine Extended Backus Naur Form syntax.

Briefly, ELC menus are made up of groups of named items. These named items may be either Terminators, or Non-terminators. Terminators represent items which can be selected. They may be used to invoke functions, or display the state of an application. A Non-terminator points to another ELC menu and its appropriate description, hence ELC menus are hierarchical. Selecting a Non-terminator causes another menu to be displayed, from which a further choice must be made. Various types of Non-terminators exist, namely Real and Virtual. Also, a Non-terminator can stipulate that at least one of its items, or sub menu items, has a true state.

ELC syntax can effectively be used to describe all types of menu structures [Apperley, M.D:1989]. The ELC syntax was implemented in Smalltalk, along with a graphical interpreter and rule set which converts the ELC syntax into an interactive menu displayed on the screen. The syntax is separate from the graphical interpreter, and the result of interacting with a displayed ELC menu is a list of Smalltalk messages which need to be sent to an Application Object. The graphical interpreter and rule set use recursive code to display and process menu interaction for the user. These can easily be modified to suit alternative interaction styles.

The ELC extensions involve the introduction of two special types of Terminators :-

Bistable Terminator.

This terminator is similar to a switch, and contains an internal state which is either true or false. When a Bistable Terminator is selected, one of two messages are returned depending upon whether the state is to be switched on or off. These messages are labelled onMsg and offMsg, and their values are defined according to the ELC description for a menu. A Bistable terminator is displayed as a menu item plus a small box which represents the state. If the state is on, the box appears black, if off it appears white, i.e. invisible - this is a simpler alternative to using a tick to represent on.

Monostable Terminator.

This terminator has no state, and only uses one message. Whenever a Monostable Terminator is selected in an ELC menu, the value of this message is returned.

ELC menus can be built by either explicitly specifying the ELC syntax, or by using the interactive menu editing tool provided. When a Linkage Slot is used for PVC menus and contains a Smalltalk message, it is expected that the attached Application Object will respond to the message with an ELC menu instance. When a Linkage Slot contains a literal, this literal must be a definition of an ELC menu using the ELC syntax. This is then executed to generate a hierarchic menu.

6.5. The Part Pluggable View Controller Tool-set.

An Application Object Direct Manipulation Interface may be constructed by explicitly defining the Part PVC description and implementing a new Smalltalk method. Alternatively, the designer may use the integrated interface design Tool-set. The Tool-set allows a Direct Manipulation interface to be constructed using a 'Design by Example' approach. This entails selecting the required Application Object, and designing the interface in real time. The Tool-set supports the selection of individual Interaction PVCs from the library of available Interaction PVC. These can then be placed in the appropriate screen positions, and sized according to requirements. Existing Interaction PVCs may also be moved around, re-sized, linked to other Interaction PVCs, or removed altogether. Existing Part PVC interfaces can be selected and added to the current interface. The interactive implementation of hierarchical menus is also supported.

The interface being implemented or modified may be tested at any time, and when completed the Tool-set automatically generates the necessary Part PVC description as a Smalltalk method code. Once generated, this PVC description can be executed by the UIMS. In doing so, the relevant user interface is reconstructed and presented to the user for interaction.

The major functions provided by the Tool-set are as follows :-

- creation, and addition of new Part Pluggable View Controllers to existing interface
- creation of the underlying interface syntax description
- addition and deletion of new Interaction Pluggable View Controllers to existing interface
- alignment and sizing of the physical windows associated with individual Pluggable View Controller View components.

- linking of Interaction Pluggable View Controllers to one another
- modification of Pluggable View Controller Linkage Slot values using an interactive inspector window
- specification of Default Part Pluggable View Controllers
- spawning of Part Pluggable View Controllers to generate new interfaces
- the interactive construction of Extended Lean Cuisine Hierarchic Menus, and the automatic generation of the appropriate ELC syntax description.

The main purpose of the Tool-set is the specification and modification of the knowledge contained within a PVC Separation Controller. As such, the Tool-set uses information concerning PVC communication requirements which are explicitly defined using external Linkage Slot descriptors.

Knowledge is also required concerning the available Application Object method protocol, or Non-Interactive Function Core. Essentially, the Tool-set assists the designer in the task of specifying which values to place in the relevant PVC Linkage Slots. The possible values depend upon the PVC external Linkage Slot descriptor, and the available Application Object methods. The Tool-set provides the interface designer with a list of possible PVC and Application Object linkage strategies. The designer then selects the required strategy, and places an Interaction PVC or Part PVC in its required position within the current interface window.

The Tool-set uses Direct Manipulation and menu style interaction, which are implemented using PVC mechanisms. Appendix H shows example interfaces implemented using the Tool-set, and the associated automatically generated Smalltalk code. The complete Tool-set is briefly documented in appendix I.

6.6. Summary.

The Tool-set and basic UIMS architecture were successfully implemented and tested on an Apple Macintosh II micro-computer, with 5 Megabytes of memory, and a 20 Megabyte hard disk. The Smalltalk system used comprises Virtual Image Version VI2.2, and Virtual Machine Version VM1.1. The system was also tested using other Smalltalk Virtual Machine platforms on the Apple Macintosh Plus, and Sun Work-station. The response time of the final UIMS is fast, and the Tool-set adequately fulfils the rudimentary

interface design requirements. Further work is needed to extend this system, and this is discussed in chapter nine.

The successful UIMS implementation has established that it is possible to completely separate the interface and application. Results from the use of the proposed UIMS have also shown that separation places certain constraints upon application and interface design. These are examined in chapter eight, which critically evaluates the proposed UIMS with emphasis given to the underlying software architecture model. This chapter also identifies the practical advantages and disadvantages arising from this approach to interface design. The implementation work is critically assessed and evaluated, and the Smalltalk 80 language is examined in relation to user interface design.

The Support of Part Hierarchy Mechanisms in an Object Oriented Language.

7.1. Introduction.

The use of object oriented languages to implement databases is a growing research interest [Tsichritzis, D.C:1988], [Lindsjorn, Y:1988], [Wiederhold, G:1986]. The support of Part Hierarchies is an issue arising out of this concern. The problem of representing part aggregation relationships in an object oriented language is examined. After initial definitions, an example Part Hierarchy is proposed and assessed according to experiences with its implementation in Smalltalk 80.

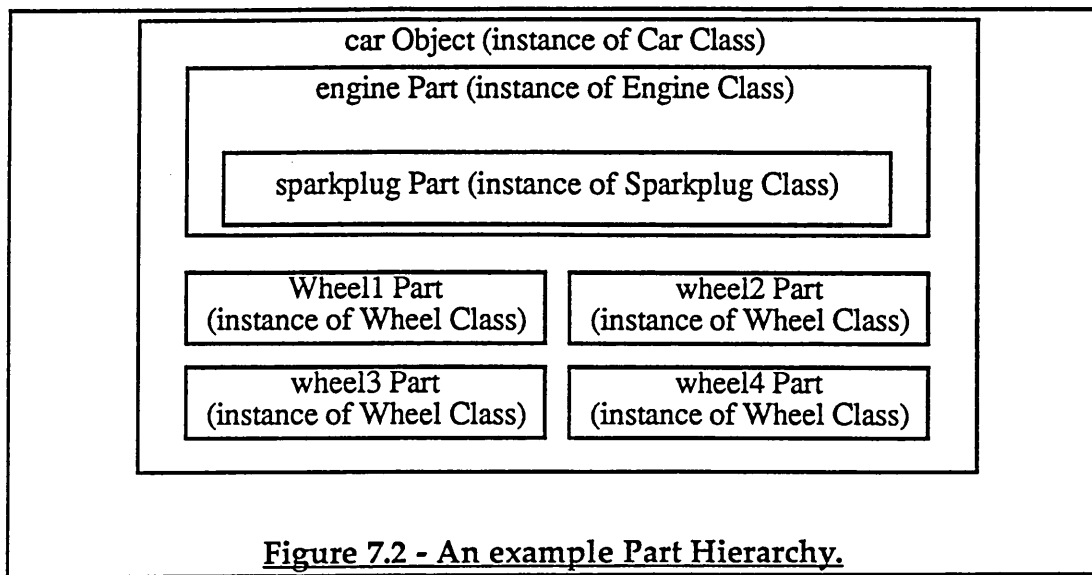
Although a diversion from the main Human Computer Interaction research objectives, this investigation of Part Hierarchies was a direct consequence of the Smalltalk problem described in section 6.4.10.

7.2. What is a Part Hierarchy, and Why is it Needed ?

In real life, objects are often constructed from parts using the whole-part relationship. A whole is made up of parts, which themselves represent further wholes, and can be made up of other parts, and so on. Individual parts belong to the whole, which can therefore be referred to as the owner of the parts. This creates a hierarchy of parts which needs to be modelled in an object oriented language. However, we are confronted with an apparent dilemma: either sacrifice the data encapsulation properties of an object oriented language, or utterly flatten the Part Hierarchy [Blake, E:1987].

The following examples, along with figure 7.2 illustrate the concept of a Part Hierarchy :-

- a Chair is made up of 4 legs, a seat and a back
- a Car is made up of a chassis, an engine, 4 wheels, etc
- an Engine is made up of an engine block, cylinders, crankshaft, and sparkplugs.



The advantages of object oriented Part Hierarchies arise from part re-use. Parts are in themselves separate objects which have their own behaviour and state, and can usually exist by themselves. As a result, the same part can be used to assemble different owner objects. For example a wheel object may be re-used in car, lorry, bike, and bus objects. This facilitates system maintenance, and effectively reduces programming effort. In many ways Part Hierarchies are the object oriented equivalent of top down step-wise refinement. They are an essential feature for object oriented languages which are to be used in modelling the real world.

The features which good Part Hierarchy implementations should provide are :-

- parts must themselves be distinguishable from the whole
- parts must be reusable within different assemblies
- information should be stored, and behaviour implemented in a Part Hierarchy at the corresponding logical level
- information about the whole must not be stored in the parts
- part information and behaviour which is not affected by the whole should remain within the part.

In summary, the whole should know about the parts, while the parts need not know about the whole. Typically, most object oriented languages do not provide facilities to describe the explicit relationships between different objects, especially the part relationship. The problems associated with the implementation of Part Hierarchies are now discussed.

7.3. Difficulties with Object Oriented Languages and Part Hierarchies.

Data abstraction is a fundamental aspect of object oriented languages. Proper object oriented data abstraction results in an explicit object interface protocol and a hidden local state. It results in object encapsulation, whereby the local state of an object can only be modified or accessed by the functions explicitly defined for it. When an object is assembled from its parts, these parts are no longer independent. A part belongs to the local state of the whole, and communication between the part and the whole is mediated by the whole, or owner.

This creates a problem. If object oriented principles are strictly followed, then the existence of parts should be invisible to the user of the whole object, as they comprise the hidden internal state. Effectively, the method protocols understood by a part may have to be implemented again in the whole. As the part may adequately implement some of these methods, this defeats the objective of Part Hierarchies. The effect of this restriction is that Part Hierarchies are replaced by a single monolithic whole as far as the external world is concerned.

Consider the example Car Class from appendix D. A Car would have to hide the engine, wheels and chassis parts within its internal state. This means that it would have to implement its own engine, wheel and chassis method protocols. Similarly, other objects which may wish to use an engine chassis or wheel would also have to implement their own part method protocols, which may be identical.

Alternatively, if parts were not hidden in the internal state of an object, then a user could explicitly modify parts. These modifications could be done without the knowledge of the whole, and the resulting changes may violate the integrity of the whole. Similarly, the response of a part could be made independent of the whole. This breaks the data hiding and encapsulation principles of object oriented languages.

Using the same example, the engine size of the engine part belonging to the Car could be modified without affecting the power of the Car. The power of a Car depends upon the engine size and total weight, and unless the engine part notifies the Car owner of a relevant change, the power can not be updated. Likewise, a wheel part could respond to the *pressure* message without considering the effect of the weight of its Car owner.

An implementation of Part Hierarchies is required which meets these data abstraction demands. At the same time conceptually correct access must be allowed to the individual parts and whole by the Part Hierarchy user.

7.4. Solving the Problem.

In summary, the implementation requirements are as follows :-

- parts need to know which changes are significant to their owner
- parts need to know what properties are affected or modified by their owner
- owners need to know what behaviour is provided by their parts.

The object dependency mechanism described in appendix B is insufficient to meet these demands, and a more complex solution is needed. Using Smalltalk 80 as an object oriented implementation language, the following Part Hierarchy concepts are presented in order to develop a final solution.

Every object which uses parts must provide individual part methods which return the actual part. For example, a Car which uses an Engine and a Chassis part must supply two methods which return the actual Engine and Chassis part. The name of these part methods also identifies the part names, for example, *engine* and *chassis*. The object returned by the part method represents the part, which must be a Smalltalk object. The part returned may be stored within the internal state of the owner, that is as an internal data field, or it may be constructed by the named method. This follows the data hiding principles of object oriented languages, where the internal state is hidden. The user of an object should not be able to determine whether the response of an object to a message is actually stored within the internal state, or constructed from the internal state.

Information processing occurs within Smalltalk as a result of sending simple messages, and associated arguments, to specific objects. A more complex type of message is required for Part Hierarchies, namely a Compound Message. This is made up of a series of simple Smalltalk messages separated by full stops, for example, *engine.sparkplug.efficiency: 20* which can be sent to a Car in order to change the efficiency of the sparkplug

part belonging to its Engine, to 20 percent. Compound messages are interpreted by a Part Hierarchy in a specific way. The last part of the message, which follows the last full stop, is known as the selector part and represents a simple Smalltalk message, i.e. *efficiency: 20*. The first part of the message, up to the last full stop, is known as the class part, and identifies the part to which this simple Smalltalk message is to be sent, i.e. *engine.sparkplug*. The class part can be broken down into individual part names, each separated by a full stop. The order from left to right identifies the Part Hierarchy, e.g. the sparkplug part of an Engine which is itself a part of the Car object.

Access to parts is always provided by the owner. Any messages, simple or Compound, sent directly to a part are automatically redirected to the part owner. If the owner is itself a part, then it also redirects the message to its owner, and so on until the top of the Part Hierarchy is reached. An owner object is therefore given the choice of modifying the behaviour of any of its parts. Similarly, an owner object can 'intercept' a message sent to one of its parts, and effect any changes which occur in itself as a result. However, an owner object may send messages directly to its parts. Such messages which pass down a Part Hierarchy are not automatically redirected, and can only arise within the internal method code of an owner part. All other messages which arise external to an object are redirected appropriately.

Message redirecting is achieved by the automatic generation of a Compound message which is sent to the part owner. Consider the message *efficiency: 20* being sent to a Sparkplug which is part of an Engine, which is itself part of a Car. The Sparkplug, knowing that it is a part of an Engine with the part name *sparkplug*, constructs the Compound message *sparkplug.efficiency: 20* and sends it to its owner Engine. Knowing that it is part of a Car, the Engine again redirects this message. The Compound message *engine.sparkplug.efficiency: 20* is now sent to the owner Car. This Car is not a part, therefore it does not redirect the message any further. A Part Hierarchy therefore requires that the relationship between owners, or wholes, and their parts is known within the system.

Individual owner objects may implement methods which override the methods implemented by their part assemblies. These overriding methods have names which take the same form as a Compound message. For example, a Car Class may implement an instance method named *engine.size:*, which overrides the normal Engine Class instance method called *size:*, whenever an Engine is implemented as part of a Car. Similarly,

a Car Class instance method called *engine.sparkplug.efficiency:* would override the *efficiency:* Sparkplug Class instance method whenever a Sparkplug is implemented as part of an Engine, which is itself implemented as part of a Car. Overriding methods may implement their own code accordingly, and if necessary may call the overridden part method.

Once redirected, a part message eventually reaches the top of a Part Hierarchy. At this point, the overriding messages may take effect. The top owner is given the first option of overriding a Compound message resulting from a simple message being sent to a part. For example, a Car checks to see whether it has an overriding method for the Compound message *engine.sparkplug.efficiency:*. If so, this is instantiated and the result returned. If no overriding message is found, the Compound message is broken down and sent to the next part in the hierarchy. In this example, the Compound message *sparkplug.efficiency:* is now sent to the engine part belonging to the Car. The process is then repeated until the simple message is sent to the lowest part in the hierarchy. That is, the engine part now checks to see whether it overrides the Compound *sparkplug.efficiency:* message, if so the appropriate method is executed. If not, the final simple *efficiency:* message is sent to the sparkplug part belonging to the engine, which itself belongs to the top most Car owner.

In summary, any messages sent by the user to a part, are redirected to the top owner in the Part Hierarchy. In doing so, a Compound message is created. If a user wishes, they may also send a Compound message to a part. Again, this is compounded further, and redirected to the top owner in the Part Hierarchy. Internal messages sent by the method code of an object are handled differently. An owner object may send a message to one of its parts by simply naming the part and sending the message. If a simple message is sent, then the part handles it appropriately, and no redirecting is performed. If the message is Compound, the immediate receiving part is given the option of overriding the message. If no overriding takes place, the Compound message is broken down and sent to the leftmost part. For example, consider a Car Class instance method which sends the message *sparkplug.efficiency* to its engine part. This message arises from an instance method implemented by the Car Class, rather than from a user. Instead of redirecting to the top most part, an engine part checks first whether it overrides this message. If so, the appropriate code is executed. If not, the Compound message is broken down, and the *efficiency:* message sent to the sparkplug part belonging to the Engine. However, when a message is sent by

a part object to itself (using the Smalltalk *self*, or *super* construction as described in appendix B) it is treated as a message which arose from the external user. As such, it is redirected to the top of the Part Hierarchy, and a suitable Compound message constructed as described above.

Overriding methods must inform dependents of the effects of part changes using the *change* or *changed:* messages. An owner may also inform dependents of changes in a part by issuing the *changed:* message with the changed part identified using a Compound message, for example *self changed: engine.sparkplug*.

Message redirecting, and the use of overriding methods should be invisible to the user. The main issue concerning the implementation of Part Hierarchies in an object oriented language is the representation of the relationships between parts and whole. This is discussed further in the next section.

7.5. A Solution Implemented in Smalltalk 80.

Compound messages are already provided for in later versions of Smalltalk 80. This provision was made to incorporate multiple inheritance within Smalltalk [Borning, A.H:1982]. The existing Compound message implementation is extended to allow Part Hierarchy Compound messages to be constructed and broken down into part and message constituents.

Smalltalk 80 provides a method called *doesNotUnderstand:*, which is used to handle errors. If a message is sent to an object whose Class or Super-classes do not implement an appropriately named method, the *doesNotUnderstand:* method is invoked with the argument set to the incorrect message name. This is used as the basis for handling Compound messages sent to part owners. If an object is sent a Compound message which it does not override, the *doesNotUnderstand:* method is invoked. The existing *doesNotUnderstand:* method code is modified so that Compound messages which are not overridden are broken down and passed onto the relevant parts. Overriding methods are implemented as normal Smalltalk methods in the appropriate Class using Compound message names.

Every object which uses part assemblies must implement a special Class method called *Parts* in its Class definition. This should return an array of

symbols each representing the name of its parts. These symbols when sent to an instance of the Class, must return the part object. This part list is not essential for the functioning of Part Hierarchies. However, it is required for the construction of Part Pluggable View Controller (Part PVC) hierarchies which match the Part Hierarchy. It is used by the interface Tool-set described in section 6.5 to list the available part interfaces. The central issue, and major difficulty, is that of representing and identifying the owner-part relationships. These relationships must be identified in order to redirect part messages to the part owners. The implementation of these relationships must be invisible to the Part Hierarchy user, and at the same time must not drastically increase the normal message handling processing time.

One implementation would be to maintain a list of owner part relationship pairs. Whenever a message is sent to an object, this list would be checked to see whether the object was a part. This places an overhead on the message interpreting mechanism of Smalltalk, although it could be reduced by building it into the Smalltalk system at the interpreter level. Part Hierarchies would only require the relationship between part and owner to be maintained, as an owner already knows about and can access its parts. Unfortunately, this solution would require the Smalltalk programmer to specifically add part relationships to this list as they are made. Similarly, part relationships must also be removed from the list when they are broken. This overhead becomes even more impractical when we consider that some parts are not actually stored in the internal state of an owner, but are constructed from it. In short, part owner relationships should be automatically maintained by the system, and not by specific programmed methods. This facility is built into the UIMS presented in chapter six. Provided a Part Hierarchy is accessed using an interface implemented with the PVC mechanism, the relationships between parts and owners are automatically controlled. When a Part PVC interface is used to interact with a part belonging to an owner, the Part PVC maintains its own representation of the underlying Part Hierarchy. Likewise, when a Part PVC is used to interact with a part which belongs to an owner which itself belongs to another object, and so on. The provision of Compound messages, and message overriding is hidden from the user of the Part PVC interface.

7.6. An Improved Solution.

The immediate problem with the proposed Part Hierarchy implementation concerns separation. The Part Hierarchy is incorporated as an integral component of the UIMS implementation. Although from a programming point of view it can be separately identified, in order to use the Part Hierarchy, the UIMS must be used. The reason for this type of implementation was to simplify the required code. The Part Hierarchy also slows down the UIMS architecture, and the deeper the Part Hierarchy, the slower the system becomes. A refined solution would implement the same Part Hierarchy as an intrinsic component of the Smalltalk system. This would have to be incorporated at a low level within the Smalltalk system, and requires extensive system redesign.

One problem not addressed by the proposed Part Hierarchy implementation is the handling of messages sent by a part to itself, or to its Super-class. Again, in order to handle this problem the refined Part Hierarchy implementation would need to be at a low level. Such a solution would also require the introduction of formal Smalltalk part creation methods. Every new object created would need a special internal data field which identifies how the object was created. In the case of new objects returned by a part method, this field would point to the owner object and identify the part name. The Smalltalk message interpreter would then need modifying to check whether a message receiver is a part. If so, the message could be redirected accordingly. Finally, the interpreter would need to distinguish between messages arising externally to an object, and those which arise from within the object method code of another object.

7.7. Summary.

Although incomplete, the proposed Part Hierarchy mechanism provides a useful addition to object oriented languages. It considerably enhances the existing data abstraction and encapsulation rules, and provides an extra facility to assist the design and implementation of object oriented information systems. Further object relationships such as associations with other objects also need to be modelled by object oriented languages. This requires further investigation, and research into Part Hierarchy mechanisms should assist this additional work.

Finally, it should be noted that the proposed Part Hierarchy implementation is conceptually different from how it is perceived by the user. Modified part behaviour is implemented by the owner. This behaviour then overrides default part behaviour, which would normally be seen if the part were used as a separate object. In the real world, the behaviour of a part is never contained in the owner. However, the Part Hierarchy mechanism appears different to the user. When this Part Hierarchy is used, it appears that the modified behaviour is actually provided by the part rather than the owner. If the implementation were to contain modified part behaviour in the actual parts, then the required code would be far more complex. This makes use of data abstraction, where the internal implementation of an object is hidden from the actual external interface.

Critical Evaluation and Assessment of the Proposed User Interface Management System and Smalltalk 80.

8.1. Introduction.

This chapter evaluates the User Interface Management System (UIMS) software architecture described in chapter six. The chapter consists of two main sections. Section 8.2 examines interface separation and evaluates this software architecture based upon experience with its implementation and application. The key issues surrounding interface separation are discussed and conclusions are drawn. Relevant criticisms and suggestions for further work are made, and these are later detailed in chapter nine. It is important to emphasize that the proposed architecture represents a research vehicle and is not intended for commercial use, although chapter nine describes various possible enhancements which would lead to the development of a professional object oriented UIMS in Smalltalk 80.

Section 8.3 evaluates and reviews the Smalltalk 80 object oriented programming language and environment. Particular attention is given to the application of the object oriented paradigm in designing 'usable' computer information systems. Several criteria influenced the choice of Smalltalk as an implementation language for the proposed UIMS. These criteria are detailed, and justifications are made to support the selection of Smalltalk in the form of comparisons to traditional imperative languages. Problems arising from the use of object oriented languages are also described, while further refinements to Smalltalk 80 are again detailed in chapter nine.

8.2. Evaluation of the Underlying User Interface Management System Software Architecture.

The potential of separation is ultimately determined by the underlying interface software architecture. A list of advantages arising from interface separation is given in sub-section 8.2.1. Sub-section 8.2.2 examines how complete interface separation was achieved within the proposed software architecture. This sub-section also discusses other interface separation approaches, and draws conclusions regarding software requirements which must be met if complete separation is to be achieved.

It was discovered that the use of software architectures based upon interface separation imposes certain design constraints. Sub-section 8.2.3 examines these constraints, and draws attention to the ways in which a separable interface can affect the application, and vice versa.

8.2.1. Advantages of Interface Separation.

The major benefits of interface separation discovered by this work were interface standardisation and consistency, component re-use, application and interface maintenance, provision of focal points for the application of specialist knowledge and research, customisation, potential improvements in Systems Analysis and Design methodologies, support of specialist interface design tools, and improvement in the 'usability' of actual user interfaces.

Different interfaces implemented using the same separable architecture can utilise identical components such as windows, buttons, and menus. This effectively enforces standards and consistency which will be seen throughout all of the available interfaces. As described in chapter two, this can improve user acceptance of a user interface.

Component re-use is also of advantage to the interface designer, it is equivalent to modularity within software engineering [Sommerville, I:1985], and bestows similar advantages. Libraries of specialist interface components, or Interaction Pluggable View Controllers can be easily maintained and expanded. Individual components from these libraries may then be tailored to suit the fine detail requirements of different interfaces, and to enable interface customisation. Savings in implementation time and cost should result and interface reliability improved. This should be particularly beneficial when complex interfaces, comprising of many different components are being developed.

Separation provides a focal point for improving interface design. Interface separation enables the practical implementation and testing of specific Human Computer Interaction theories and technology. This should provide a clearly defined field of research, whose body of knowledge can be distinguished from other computer science research endeavours.

Interface customisation for individual users, either by the users themselves or through an Intelligent Interface, can help solve the problem of designing interfaces for a user population with varying needs. Separation draws a clear line between interface and application, thus providing a framework for customisation. The application should not be customised, and the line drawn by separable architectures protects the application functionality from any customisation.

Personalized interfaces may be implemented for individual users, user groups, or complete organisations. This work has shown that many different interfaces can be implemented for the same set of application functions. Customisation can be performed relatively easily using specialised tools associated with separable interface architectures. This should reduce interface development cost and provide a great deal of design flexibility. Although it is unlikely that exactly the same interface can be re-used with different applications, it should be possible for various interface preferences, such as style, to be transferable between different interfaces.

Finally, separation has the potential to improve the complete Systems Analysis and Design process. Using separation, it should be possible to independently design, implement and test the interface and application. New user centred design approaches can be developed, and specialised knowledge and tools can then be applied. This can be equated to the effect of formal software engineering principles upon conventional computer software development.

8.2.2. Software Requirements for Interface Separation.

This work has provided a practical investigation into interface separation. A more formal abstract approach to separation was taken by Cockton [Cockton, G:1986], who has arrived at similar theoretical conclusions. In summary, Cockton presents a definition of separation based upon operational semantics, and the explicit representation of the state vector in the state to state transition relation between user interface and application. He concludes that a system described using a transition relation has separable sub-systems, if :-

- there is partition of the rule set,
- there is partition of the state vector object set,

- there is a bijective mapping from rule sub-sets to state object sub-sets,
- each rule in a sub-set is exclusively triggered by references, and changes only those state objects in the sub-set associated with it by the bijection.

The proposed UIMS has provided a practical insight into the requirements, and constraints of interface separation. These requirements and constraints are now discussed.

8.2.2.1. The Number of Components required.

The proposed PVC software architecture uses three components, namely: Application Function Set, Separation Controller, and User Interface. The interface is itself comprised of an Interaction Controller and View component. However, this refinement is intended to improve actual interface design and does not affect the overall application and interface separation. The three component architecture is shown in figure 8.2.2.1 (b), where it is contrasted with a two component system (a) comprised of a User Interface and Application Function Set.

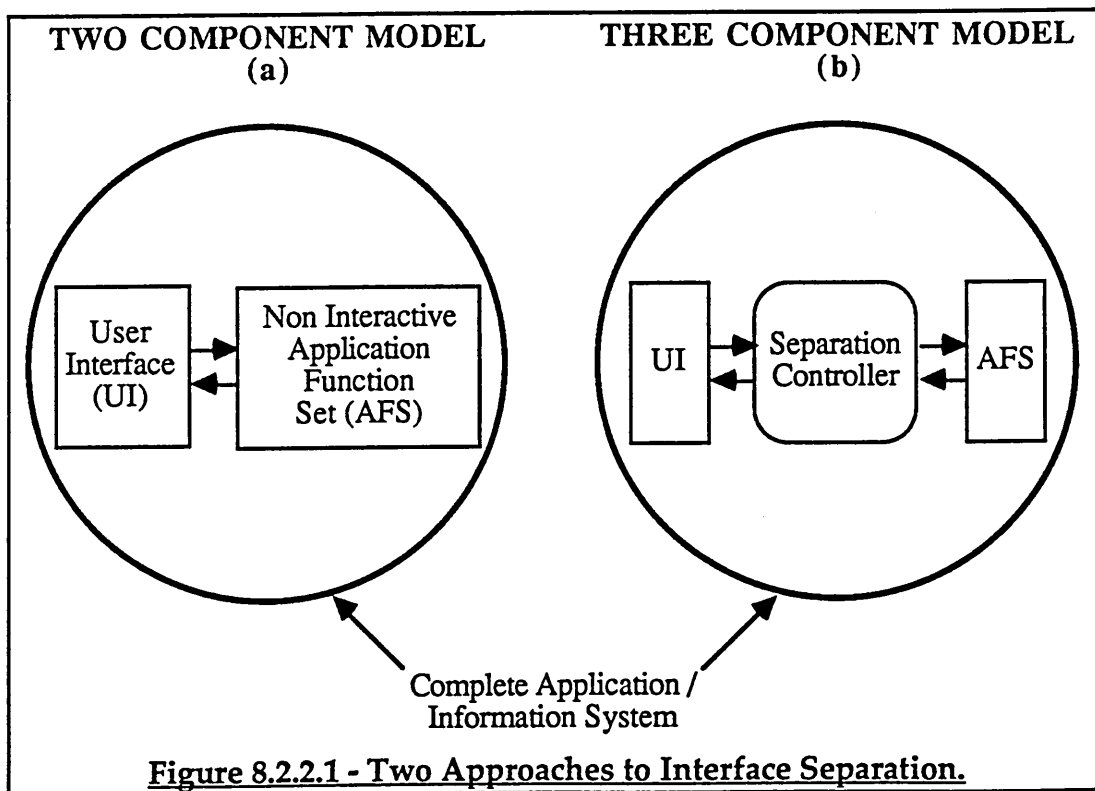


Figure 8.2.2.1 - Two Approaches to Interface Separation.

A two component system requires a compromise whereby either the interface knows about the application (1), or the application knows about the

user interface (2). Compromise (1) results in an interface which directly calls application functions, and an example would be the Model View Controller mechanism described earlier. The interface must know about the application function names and preconditions. As a result it is harder to re-use different components of an interface, and perform any customisation. Compromise (2) is often seen within the 'tool kit' approach, such as that of the Graphics Environment Manager system. The interface is modularised into components, but the high level interface structure is threaded through the application functions. The application functions use different interface components to fulfil their user interaction requirements, and interaction control remains primarily within the application. Again, this makes it difficult to customise interfaces, and experiment with different dialogue control mechanisms.

For strict separation to be achieved the interface state transition rules cannot directly reference or be directly conditional upon the state transition rules of the application functions, and vice versa. Mutual ignorance is only achievable if the two components have no direct access to one another's state information and related data. That is, no communication is allowed between the two, and a third intermediary component is required.

Separable components are separate systems which cannot be sub-systems of a super-system without the use of an intermediary component. This strict separability is compromised within the two component system. Only a three component system suffices. The extra intermediary component 'knows about' both the two separate components, and controls or coordinates communication between them. With respect to the proposed architecture, this intermediary component represents the Separation Controller, and the two separated components are the User Interface and the Application Function Set implemented by the Application Object.

The Separation Controller is critical to any User Interface separation model. It enables dialogue control to be contained within the interface. At the same time it may allow application preconditions to be specified by the application function set. These preconditions permit application functions to be blocked due to unavailable data. When the interface makes this data available, the relevant functions can then be instantiated and the results returned. An application may state its preconditions, but does not specify what user interactions are used to meet them. The Separation Controller contains this knowledge, which may vary between different interfaces. For example, an

application function may specify that it requires 10 integer values, and a boolean 'true' value before it can be invoked. These preconditions may then be mapped onto interface processes by the Separation Controller. Similarly, the Separation Controller may also add its own preconditions which are independent of both the application and interface. These can be varied without affecting the interface or application. Strict separation therefore requires a strategy for instantiating the interface processes and application functions in a way that avoids mutual dependencies. This strategy must therefore constrain any design decisions. These constraints are discussed below.

8.2.2.2. Application Function Set and its Preconditions.

The proposed architecture deals with all possible types of interface and application communication, apart from Active Dialogue which is initiated by the application. Figure 6.4.2b in chapter six summarises the four possible types of interface and application communication. It also shows how this architecture supports Input, Output, and Passive Dialogue in relation to the application.

Active dialogue occurs when an application function requests specific information from the user through a user interface. In effect, the application interrupts and takes over the interaction dialogue control. Whenever this occurs, knowledge concerning the interaction dialogue is also required. That is, how the information is to be collected.

The application function may itself specify how to collect the information. For example, 'ask the user to confirm deletion using a button', or 'using a text editor'. This results in the compromise two component system described above, wherein the application specifies interface dialogue control. Alternatively, defaults may be specified. For example, a certain interface may specify that all requests for boolean values appear as buttons, and all text requests use a special type of editor. This type of generalisation is again unacceptable, as it severely reduces the flexibility of a separable architecture. Another technique would be to explicitly define any application interaction requirements. Whenever a new interface is to be implemented, the designer would specify what interface processes should be used to meet these requirements. This would be impractical, especially where large applications are concerned. Even if a particular application function were not used, the

interface designer would have to provide this link. Again, this would also give dialogue control to the application.

In conclusion, Active Dialogue should not be allowed if strict separation is required. However, the ability for application functions to specify data requirements must be provided. This can be fulfilled using application function preconditions. The proposed architecture does not support application preconditions, as this would require a parallel implementation and further extensive work. Nevertheless, the proposed architecture has provided a useful insight into this requirement.

An analogy can be made to Batch Systems, which cannot interactively acquire data and are typically controlled by a Job Control Language. Any data requirements must be met by including the data in the batch function call, or by supplying a suitable file. Batch functions may call other batch functions and may communicate using shared files. Individual batch functions may also be partially invoked and therefore made to wait upon incoming data. For example, a sort function may be invoked which reads and sorts records until some terminator record is given. The function does not know how many records to sort, and may be able to sort records from many different sources. If a parallel architecture is available, batch systems may also allow several suspended asynchronous functions, each of which awaits certain preconditions.

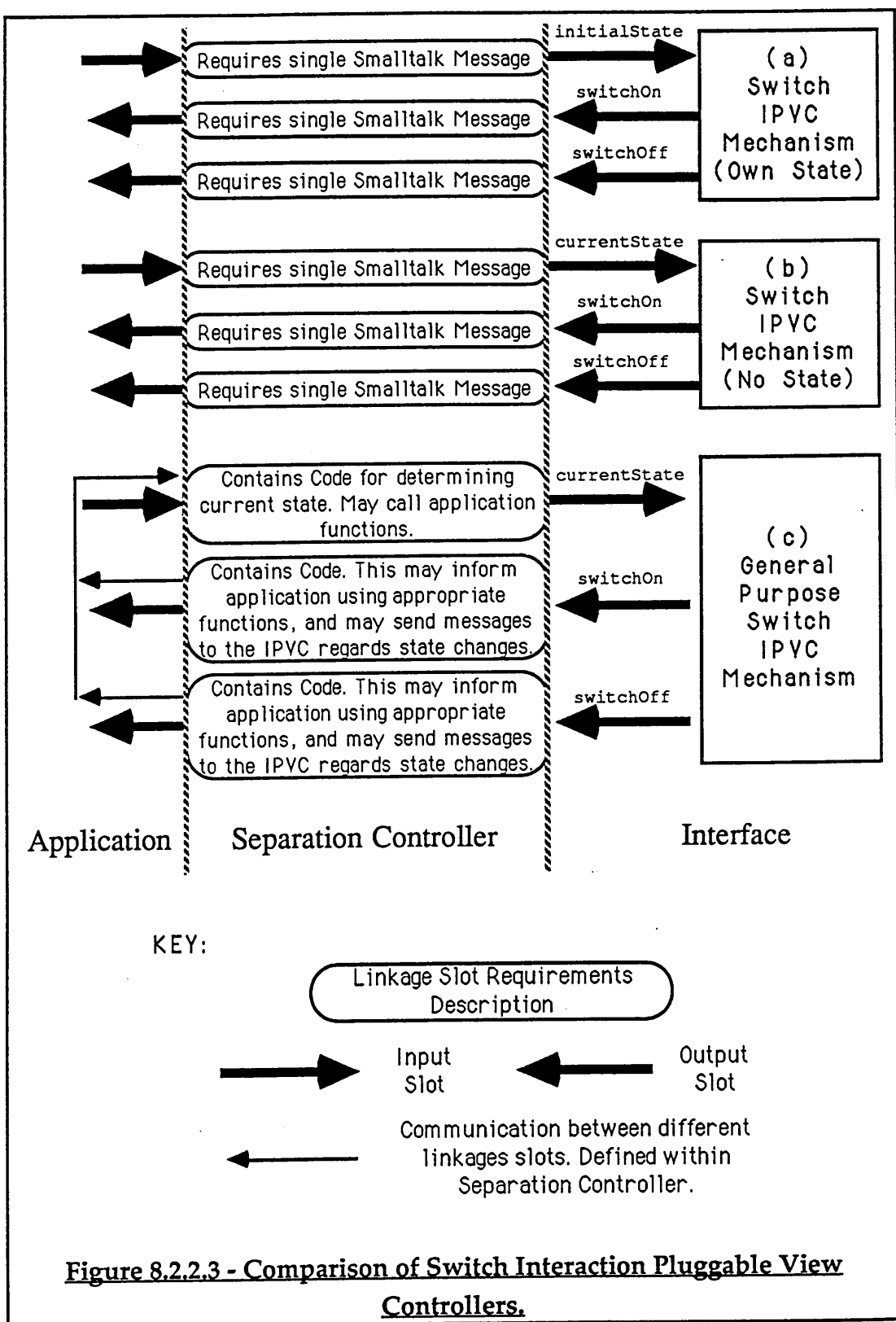
Application function sets should be designed and implemented as batch systems. They may however, specify function preconditions. For example, an averaging function may specify that it requires ten numbers as its argument list. The Separation Controller can ensure that these ten numbers are provided before the function is invoked. The application does not specify how these numbers are to be input, and this is left to the interface designer. Another example would be an application function to delete a record which requires a record key and boolean confirmation. The Separation Controller will not invoke the function until both preconditions are met, and it is up to the interface designer to decide how this is mapped onto user interactions. The complexity of these preconditions depends upon the implementation, and a formal language is needed for their specification.

8.2.2.3. Application and Interface States.

An application has many states which may be represented within its interface. At any time, the application will always be in one of these states. The current state changes as a result of user interactions defined by the interface processes, and may also be affected by other external influences. An interface also has its own set of states.

Consider the example of a Switch Interaction Pluggable View Controller (Interaction PVC) as shown in figure 8.2.2.3. This Switch has a state of either on or off, and instantiates one of two Application Object functions depending upon whether it is switched on or off. The Switch state may be maintained within the Application Object (b), or within the Switch itself (a). Where the state is maintained by the Application Object, the Switch is linked to the Application Object using a specific function which returns the state - using a Linkage Slot called *currentState*. When the state is maintained within the Switch, no such link is required. Instead an Application Object function link is needed which determines the initial state - using a Linkage Slot called *initialState*. After this is determined, the Switch maintains its state separate from the Application Object, according to user interactions. Two different types of Switch are therefore required.

When the Switch maintains its own state (a), this state may map directly onto an Application Object state. As a result, it may be possible for the Switch state to be out of phase with the equivalent Application Object state. This is because the Switch and Application Object states are only synchronised at the beginning of an interaction session. If this is to be avoided, the Application Object must update its state correctly whenever the switch on, and switch off messages are issued, using the values contained in the *switchOn* and *switchOff* Linkage Slots.



Where possible, PVCs should not maintain separate copies of Application Object states. These should be derived directly from the Application Object. Whenever user interaction occurs, the appropriate Application Object functions should be instantiated. The PVC should not change its state as a result. Instead it should wait for the Application Object to inform it of

appropriate state changes. Using the Switch example, user interaction resulting in switching on or off should invoke the Application Object functions defined within the Separation Controller. The Switch PVC View should then wait for the Application Object to inform the Separation Controller that its state has changed in a way which affects the Switch PVC View representation. The Switch PVC View can then update itself accordingly for example, by the use of screen highlighting.

The problem of states is further complicated by situations where the state of a PVC does not have a direct mapping onto an equivalent Application Object state. Consider an Application Object which represents a TV. This may provide a function which answers queries regarding the current channel, and a function which allows the channel to be changed. A particular interface may implement a bank of channel selector Switches labelled according to the channel. These Switches are linked to one another, in that switching one on has the effect of switching all the others off. For each Switch, the required switching off Application Object function is set to nil, while the switching on function is set to the Application Object channel change function, along with the necessary argument e.g. 'BBC1', 'BBC2'. The state of individual Switches is determined using the Application Object function which enquires on the current channel. The result from this is then compared to the channel name, and if it matches that of the Switch, the Switch is set on, otherwise it is set off. Here the Switch state is indirectly mapped onto an Application Object function. The existing architecture cannot cope with this situation, as the simple Linkage Slots are insufficient.

As shown in figure 8.2.2.3 (c), Linkage Slots should be allowed to contain actual Linkage programs, which may take the form of Smalltalk code. These programs would be contained within the Separation Controller, and issue appropriate function calls to the Application Object. A Switch Interaction PVC would still specify that two output (switching off and on), and one input (determining switch state) Linkage Slots are required. However, these slots could contain actual code for comparing results of Application Object functions. This would also enable application preconditions to be handled, and linked to interface processes.

Linkage Slots should also be permitted to send messages to the PVC itself, or other active PVCs; bearing in mind that an PVC only receives messages from the Separation Controller, and does not need to know where the messages originated. As a result, a single Switch Interaction PVC (c) could be

implemented to cope with both types of Switch described. When the Switch state is maintained within the Application Object, the state Linkage Slot would contain the Application Object function name which returns the current state. When it is maintained within the Switch, this Linkage Slot would contain code which sends a message to itself informing of the new Switch state. Although it appears that the Switch maintains its own state, this code is contained within the Separation Controller and is not specified within the implementation of the Switch Interaction PVC. This removes the need for the provision of communication between Interaction PVCs discussed in section 6.4.6. Finally, the on and off Linkage Slots of a Switch may be set to nil, effectively enabling a PVC to be used without an actual Application Object being present. This provides a flexible means of interface prototyping.

Higher level Application Object functions may also be built within the Separation Controller. If a new function can be built by the structured combination of existing Application Object functions, then this may be accomplished by implementing Linkage Slot code which invokes the necessary Application Object functions. This is an alternative to implementing new Application Object functions, which may only be required within one customised interface.

8.2.2.4. Component Communication Requirements.

Associated with the Separation Controller must be a formal message, or token passing mechanism. The Separation Controller receives messages from, and sends messages to both the interface and the Application Object. Different Separation Controller implementations may define different formal definitions. In the case of this implementation only two types of messages may be sent by the PVC View or Interaction Controller. These are either requests for Application Object states or information, or messages informing of particular user interactions. Accordingly, the Separation Controller issues two types of message to the Application Object either asking for specific information, or informing of user interactions. Two types of messages can be received from the Application Object. These are either a message informing of a state change, or a reply to a request for information. The state change informant message takes the form of a simple *changed* or *changed: argument* message, where argument identifies the Application Object function whose response has altered. The reply takes the form of a Smalltalk object. Finally, two types of message can be sent by the Separation

Controller to the PVC. These are either messages informing of Application Object changes, or messages returning a value in reply to a query made by a PVC.

The actual messages which are sent depend upon the knowledge contained within the Separation Controller. Individual PVC Separation Controllers decide whether the related PVC View is affected by any Application Object state changes. It then issues the appropriate messages to the View. The proposed architecture only supports synchronous communication. Requests by a PVC for information causes the interface to pause until a reply is received from the Separation Controller. A need for asynchronous parallel communication was identified, to enable the interface and Application Object to function independently. While testing the architecture, the need for further message passing requirements was also recognised. These are now discussed.

Based on experience with the library system, indicators are needed within the interface to show what percentage of a particular task has been completed. Whenever a lengthy task is instantiated by the user, the Separation Controller must monitor the completion of an Application Object function, and inform the PVC accordingly. This requires that the Application Object provides functions which allow the Separation Controller to enquire on the percentage completion of a task. When required, a PVC must also provide facilities to receive messages from the Separation Controller relating to the percentage completion of a task. The Separation Controller must then contain knowledge which controls and instantiates the completion monitoring task. Suitable information can then be presented through the interface for the user.

Error handling was not considered during the design of the architecture. However, a need was identified for messages which deal with application errors. Theoretically, the only type of application errors which may occur are due to resource failures such as disk errors. Errors due to incorrect function instantiation, and incorrect or invalid user interaction, should not affect the Application Object if it is consistently designed, and the interface is correctly defined upon it. Provision must therefore be made for error messages to be sent by the Application Object to the Separation Controller, informing of any resource failures. The Separation Controller can then inform the Error Handling module, which can then take necessary action such as displaying

alert messages, and on-line assistance. Recovery strategies may also be required and may need to be defined within the Separation Controller.

Interaction errors within the interface are of a different nature, and should again be managed by the Error Handling module. Errors such as misspelling, and invalid interactions, should not be passed onto the Application Object. Note that help and tutorial information concerning the Application Object should be maintained separate to the Application Object, and made available to the appropriate Intelligent Error Handling sub-system.

Also relating to errors, further messages are required for undoing and redoing Application Object functions. These messages could be sent by the interface, and need to be linked either directly to suitable Application Object functions, or to high level functions implemented within the Separation Controller.

8.2.2.5. Separation Controller.

The Separation Controller knows about both interface communication requirements, and Application Object function requirements. These are then mapped onto one another using information stored within this component. The proposed architecture only supports a simple binary relationship between the interface and Application Object. As described, this is insufficient and the facility to include specialised linkage code, or strategies, is needed.

The Separation Controller manages communication between interface and Application Object. It must support both synchronous and asynchronous communication. It must also allow Application Object function preconditions to exist, and enable partial instantiation of certain functions.

A parallel software architecture is essential, if the full potential of separation is to be realised. Many suspended application functions may coexist, awaiting the satisfaction of their preconditions. A PVC must not be forced to wait for replies to its Application Object communication requests, although this synchronization may sometimes be required. The interface must also permit several PVC dialogue control sequences to execute in parallel, which may or may not be dependent upon each other. Suitable parallel controls are also needed to prevent incidents such as deadlock.

8.2.2.6. Interface Defaults.

Current PVC communication defaults are implemented within the actual interface components. For example default window size, default Linkage Slot values, and default text style. The actual default values should be specifically adjustable for individual interface implementations. A PVC could explicitly define what defaults are required. The Separation Controller can then store the actual default values, which may differ between implementations.

8.2.3. Interface Separation Design Constraints.

Interface separation imposes certain interface and application design constraints. Although these may restrict the interface and application designer, they are not necessarily detrimental to the design process. Like most formal methods, restriction has the potential to improve standardisation, maintainability, and quality control. The following separation constraints were discovered.

8.2.3.1. Application Independence.

Interface separation does not imply application or interface independence. Although 'physical' software separation is desirable, conceptual separation should be a matter of choice. It is up to the designer to decide what influence the application objects and operations have upon the user interface. Good interface metaphor communicates the underlying application functions, if its features suggests those of the underlying application [Rosenberg, J:1983]. Application concepts must therefore be properly understood and represented within the interface. Interface separation minimises software dependency, allowing designers to determine any conceptual dependency for themselves.

The interface presents the application to the user, and is therefore affected by the underlying application functional structure. Different application functional structures will undoubtedly require different interface structures. It is not the purpose of interface separation to provide a generic interface. In fact, this is probably impossible to achieve. It is the component parts of an interface that are generic, that is the individual PVCs, along with any interface design tools.

Some dependencies are unavoidable, such as numeric and text values, which can only be represented using certain interface components. However, it should be possible to implement special Separation Control code. This may convert the values returned by an application function, for example special code for changing text into numbers so that they can be displayed using a Bar Chart.

8.2.3.2. General Application Design Constraints.

The chief constraint is that an application must be designed as a set of Non-Interactive Functions. Consideration should not be given to user interactions, dialogue control, or graphic and textual representation. Functions may specify preconditions, but may not contain direct requests for user input, or screen output. This approach to program design is simpler to achieve using declarative and object oriented implementation languages. Where other languages are used, the programmer must constrain their implementation techniques.

Finally, application functions must inform of state changes which may affect an interface. Any implementation language must therefore provide special statements to support this requirement. For example, the Smalltalk *changed* and *changed:* messages. Depending upon the final software architecture implementation, these statements are then handled by the Separation Controller.

8.2.3.3. General Interface Design Constraints.

The interface must provide its own dialogue control. With this implementation, this was implicit within individual Interaction PVC interface components. However, it would be better to make this control more explicit thus enabling it to be easily modified. The interface can be designed, tested and implemented separate to the application. Although knowledge concerning application concepts is necessary for the interface designer, design may progress while the application is being independently implemented.

8.2.3.4. Constraints imposed by the Separate Application on the User Interface.

Constraints imposed by the application on the interface are mainly representational. These constraints are difficult to avoid. For example, an object oriented application cannot be used with a relational style interface. Concealment or misrepresentation of the underlying Application Model could result in 'deceptive' interfaces which would probably confuse the user. This must be avoided, and in most cases would probably be impossible to accomplish due to these representational constraints. However, knowledge contained in the Separation Controller may still be used to change how an application is presented to the interface. This provides flexibility, allowing for example, equivalent Application Functional Models to use the same types of interface.

Finally, the application requires that the interface can receive suitable error messages informing of resource failures within itself. The interface must handle these errors, and advise the user accordingly.

8.2.3.5. User Interface Constraints imposed on the Application.

Two main interface features constrain application design, progress reporting, and undoing. The application should provide functions which correctly undo the effect of its normal functions. A minimum requirement would be to provide an 'undo last action' function which would return the application to its previous state. Similarly, the application must also provide specialised functions which report on the progression of normal functions. These can then be used by the Separation Controller to keep the interface informed of task progression.

8.2.3.6. Summary.

Figure 8.2.3.6 summarises the dependencies that exist between interface and application. It lists the different components which may be affected due to specific application and interface modifications.

<u>Modifications and Extensions to User Interface</u>	<u>Components Affected</u>
Representational Changes	-
Changes to Dialogue Control	-
Defaults	Separation Component.
New Input / Output requirements	Separation Component.
Undoing	Separation Component, Application.
Percent Done Indicators	Separation Component, Application.
New Functions required	Separation Component, Application ?.

<u>Modifications and Extensions to Application Functions</u>	<u>Components Affected</u>
Changed Input / Output requirements	Separation Component.
Changed Preconditions	Separation Component.
Changed Error states	Separation Component, Interface ?

Figure 8.2.3.6 - Component Stability Within Separable Architecture.

A hyphen indicates that no other component is affected, while a question mark signifies that a component may be affected depending upon the extent of the modification.

8.2.4. Conclusions.

Separation is an enabling technology, providing new software architectures which offer many benefits to the system designer. Unfortunately, it is difficult to impose design methods and standards using software architectures and support tools. It is therefore still possible to design poor interfaces using separation.

Further development of improved separation architectures and tools should establish separation within the software engineering community. The full effects of separation will depend upon how it is used within new integrated User Interface Management Systems. Hopefully, it will simplify the interface design process, and enable ordinary users to implement their own customised interfaces.

Although incomplete, the proposed three component architecture should help settle the argument as to what actually constitutes an interface and an

application. The line drawn by the Separation Controller clearly determines the role and content of the separate interface and application. Although further work may change the specified interface and application boundaries, a reference model is now established.

Finally, new interface technology should be viewed in context of the complete information system design process. A poor user interface can spoil a fine application, but a wonderful interface is unlikely to broaden the scope of a narrowly defined application [Sproull, R.F:1983]. New ways of accessing the same application functions are provided. However, if these functions fail to meet the application requirements, the complete computer system will fail. Separation should help distinguish between poor application functional design and poor interface design. As such, responsibility for different aspects of computer system design can be clearly established and allocated to appropriate experts.

8.3. Evaluation of the Object Oriented Paradigm and Smalltalk 80 Programming Language.

Many people who have experience with computers find object oriented systems strange. In contrast, many people who have no idea how computers work find the idea of object oriented systems quite natural [Robson, D:1981]. Although computer experts have the experience to understand and use object oriented design tools and methods, the systems which result are often far from object oriented. The initial learning curve for computer experts is steep, and is probably due to well established preconceived ideas concerning traditional imperative programming languages. Meanwhile, computer naive people find object oriented systems easy to use, but because of their inexperience, difficult to design and implement. The comprehensibility realised by non-computer experts is most likely due to the virtues of the fundamental underlying concepts of object oriented applications, and the close relationship of those virtues to the way in which people actually view the real world. As the majority of users are not computer experts, understandable object oriented applications provide a good foundation on which to build 'usable' user interfaces, and ultimately 'usable' computer systems.

Object oriented systems directly support good interface design in several ways. First and foremost, the concept of separation is clearly supported. An object oriented application is implemented as a group of distinct objects

which can communicate by sending messages to one another. Each object provides an explicit set of 'external' methods which can be invoked by the user, or other objects. These methods therefore define the behaviour of an object, while its implementation is hidden internally within the object. The basic user interface is one of message passing, whereby the user sends a named message to an object, along with any arguments. The receiving object then executes the associated method, and returns a value depending upon the method code. As a result, it is easy to implement application objects which have no element of user interaction. Instead, each object expects to be given the necessary arguments as part of the invoking message, and therefore does not need to request any user input. Similarly, returned values are themselves objects which can be used or displayed in a variety of ways which is of no concern to the application. As this research has shown, a separate user interface can be readily implemented on top of this message passing mechanism. Unfortunately, it is still possible to implement non-object oriented applications in an object oriented language, and object oriented applications which have embedded user interaction control. However, with good training and experience this possibility can be reduced.

Intrinsic object oriented separation mechanisms can be contrasted with the mechanisms provided by traditional imperative languages. Imperative languages are based upon the concept of separate data and functions. The functions modify the data, and the results of a programs execution are usually left in the data. Encapsulation, or data hiding, is rarely directly supported and specialist programming techniques are necessary for its accomplishment [Bell, D:1987]. The concept of separation is not intrinsically supported by imperative languages, although the similar software engineering goal of modularity is provided by some languages such as Modula-2 [Welsh, J:1987]. Effectively, user interaction processes must be implemented as part of the program code and suitable language statements are normally provided. The result is a program which 'drives user input and output', rather than the object oriented situation where user input and output 'drives the program'. As this work has shown, the latter situation is preferable and necessary if true interface separation, with all its benefits, is to be achieved.

The object oriented notion of code re-use provides many benefits to the interface designer [Ingalls, D.H.H:1981]. Code re-use is supported in Smalltalk by the Class / instance relationship, whereby one Class may have many instances, each of which behaves identically but has its own internal

state. This enables the development of pre-written interface routines, or Classes which can be re-used within different interfaces. For example, standard buttons, menus, and windows. Object oriented systems, such as Smalltalk 80, which support the concept of inheritance [Halbert, D.C:1987], also enable new Classes to be implemented as Sub-classes of existing Classes. Sub-classes inherit the behaviour and internal state of their Super-classes, and add new or modify existing behaviour and state. Effectively, Super-classes represent generalisation, while Sub-classes implement specialisation. Object oriented programming typically becomes a process of programming by the modification and extension of existing facilities. Code re-use reduces the programming effort required to implement new interface components, and also improves the consistency of the interface. This increase in consistency arises out of the repeated use of identical interface components and styles, improving the 'usability' of the final interface. The proposed UIMS utilises these code re-use mechanisms, and facilitates the management of a library of Interaction PVCs and Special Part PVCs.

The programming environment provided by Smalltalk 80 also assists and enhances interface design. Interface design is primarily an interactive process. The specification of user interfaces using a specialised syntax description has its place as a method for defining and storing existing interfaces. However, it does not enable the designer to visualise the interface, and test its functioning. Interface syntax descriptions require the usual edit and compile development cycle, which can become laborious and time consuming. Smalltalk 80, and in particular the proposed UIMS Tool-set, enables interfaces to be interactively designed and tested. At the same time, the Tool-set provides an underlying interface syntax as a means of storing existing interface descriptions. The interactive environment approach to interface and application design also promotes the use of prototyping. As discussed in chapter four, prototyping assists the design and implementation of 'usable' computer systems.

Many other advantages arise out of the use of object oriented design methods and programming languages. Typically, these are associated with software engineering and relate to areas such as integrated program support environments [Barstow, D.R:1984], [Wasserman, A.I:1982], programmer productivity [Boehm, B.W:1987], and program modularity [Bell, D:1987].

Several problems also emanate from the use of object oriented methods. Fortunately these do not directly relate to interface design, and are currently

addressed within other research fields. Object oriented systems are typically large, often comprising of thousands of independent objects. This raises the issues of object persistence, with objects needing to be stored between different sessions. Due to memory constraints, these objects also need to be frequently passed between disk and memory. The underlying model of object oriented systems is difficult to match with traditional Von Neumann hardware architectures [Pountain, D:1988]. As a Consequence, there is a need for complex time consuming algorithms to accomplish the necessary paging and object management [Lambie, K:1989]. As a result, object oriented systems require large work-stations, or powerful microcomputers to run on. Hence, the choice of Apple Macintosh II hardware, with 5 megabytes of memory and a 20 megabyte hard disk drive. Current research trends indicate a move towards new hardware architectures which are specifically designed to provide 'hard wired' object management facilities. These new architectures are radically different from Von Neumann style computers, and potentially provide a hundred fold increase in the execution time of object oriented applications. An example hardware architecture is Rekursiv [Harland, D.M:1989], which should improve the viability of object oriented systems in the commercial sector.

The need for object oriented databases which provide a 'natural' model of real world objects is a growing field of research [Wiederhold, G:1986], [Tsichritzis, D.C:1988], [Lindsjorn, Y:1988]. Such databases hypothetically enable many thousands of objects to be easily managed, and provide the facility to relate different objects to one another using aggregation or association relationships. Such systems should also assist in the provision of multiple user interfaces for a single application. However, many problems arise chiefly due to the conflict between the object oriented concept of object independence, and the database concept of data independence. Again, current research is addressing these issues [Banerjee, J:1987].

A final problem relates to the controversy between static and dynamic typing in computer programming languages [Bell, D:1987]. Static typing requires that the value of a variable data item be of a specific type, which is fixed during compile time. In contrast, dynamic typing allows a variable data item to change the type of its value at any time during a programs execution. Dynamic typing is more flexible than static typing, and is provided by most languages which use interpreters and run time environments, for example Smalltalk 80, and LISP. However, dynamic typing requires that type checking is performed at run time. For example, it is meaningless and incorrect to add

a variable of type integer to a variable of type character. A compiler using static type checking would detect most errors of this sort during compilation, and would therefore not need to use type checking algorithms while a program was running. However, a language which provides static type checking could not check for this error at compile time, because the type associated with a variable may change during the programs execution. Instead type checking must be performed at run time, which requires an overhead of extra type checking code.

The problem of type checking particularly effects the efficiency of a UIMS. As discussed above, an interface is representationally constrained by the application. That is, certain types of information can only be displayed or collected in certain ways. Where static typing is employed, type checks to prevent the incorrect representation or collection of application data can be made while an interface is being interactively designed. Where dynamic typing is employed, these checks must be made at run time whenever a particular interface is used. The overhead of these checks was alleviated in the proposed architecture by assuming that an application enforced its own static types. Because Smalltalk 80 supports dynamic typing, the failure of an application to adhere to this assumption results in an error.

In conclusion, the object oriented paradigm allows for the construction of highly interactive user interfaces. It naturally leads the way to the definition of UIMS where the user interface is implemented as separable small manageable units, rather than a single monolithic system. In addition it makes it possible to distribute the semantic, syntactic, and lexical levels of user interaction at various levels of interface abstraction rather than at a single level. It also facilitates the interactive prototyping of consistent interfaces based upon the use of interface component libraries. Finally the concepts of object oriented systems are easily understood by non-computer experts, enabling the novice user to quickly understand how new applications work.

Chapter Nine.

Suggested Further Work.

9.1. Introduction.

Further work can be directed along three major avenues :-

- (a) Extending the existing implementation
- (b) Investigation of related Human Computer Interaction work
- (c) Refinement, and 'polishing' of the proposed User Interface Management System Implementation (UIMS) to produce a complete interface design Tool-set for Smalltalk 80.

Extensions (a) and (b) will be of benefit to the Human Computer Interaction research community, while (c) will be of interest to the commercial sector. As Smalltalk does not currently provide an interface design Tool-set, this work may potentially form the basis of a standard interface Tool-set for Smalltalk 80, and possibly for other object oriented languages.

This chapter summarises the further work which may be carried out as an extension to the research presented.

9.2. Extensions to existing User Interface Management System Implementation.

The primary objective for the proposed UIMS implementation was to investigate the potential of interface separation. A lesser consideration was the implementation of a complete UIMS Tool-set. As such, the implementation lacks many useful features which could make it more acceptable as a professional interface design Tool-set. This may be the subject of further work, and the following improvements are suggested.

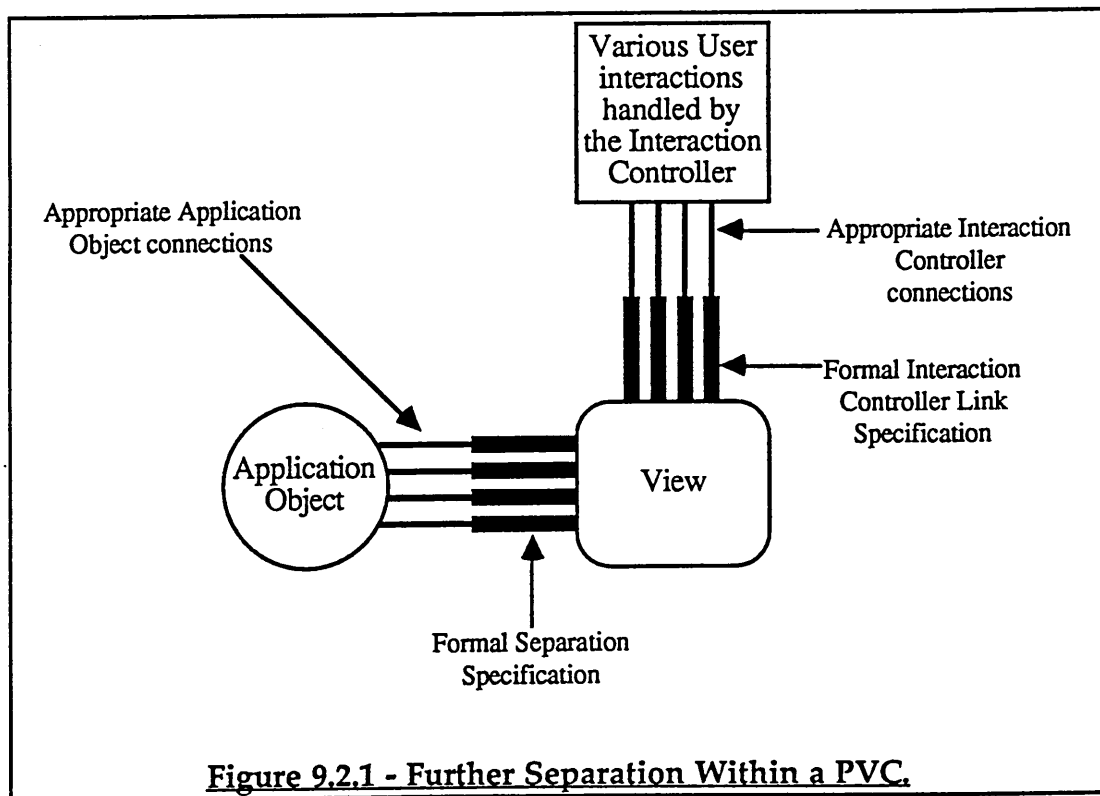
9.2.1. The User Interface Management System Architecture and Tool-Set.

Further Interaction Pluggable View Controllers (Interaction PVC) need to be implemented, covering a wider range of user interface needs. These could include Tree Browsers, further Knobs, Buttons, Sliders, Thermometer Scales, and Dials. Further Special Part Pluggable View Controllers (Part PVC) are also required, providing interfaces to various Smalltalk data structures

such as Dictionaries, Linked Lists, Bags, and Ordered Collections. Finally, new Class Browsers could be implemented as Special Part PVCs to replace the existing Class Browser interface.

The syntax for the External Linkage Slot Description, Extended Lean Cuisine Menu Definition, and Part PVC Description could be improved. A more formal language is needed to replace the existing token based language, which simply encompasses the fundamental syntactic requirements. Parsers could then be implemented to analyse the language and perform necessary error handling and automatic error correction. Although new features would not be added to the existing syntax, a formal language would be more flexible and easier to read.

The Pluggable View Controller (PVC) mechanism currently specifies an external description which defines the separation requirements of both its View and Interaction Controller. As figure 9.2.1 shows, it is envisaged that the View and Interaction Controller will eventually be completely separated, with an explicit formal description linking the two.



The formal separation specification would still define the separation requirements to be met by the Application Object. Similar to the existing Linkage Slot descriptions, this would define the input and output allowed. The Interaction Controller link specification would define the interactions

required by the View. The Interaction Controller can then map relevant user interactions onto this specification. For example, a red, blue, or yellow mouse button press can be used to trigger off the same View function. Alternatively a keyboard character can be used, or a combination of the mouse and keyboard interactions. The new PVC mechanism would be more flexible, and an extra key-stroke, or physical interaction layer added.

The interface design Tool-set itself could be refined, with more editing functions added. For example, functions are needed for :-

- PVC duplication
- PVC window rotation, and translation functions
- making PVC windows transparent, thus allowing PVCs to overlap one another
- gridding, enabling PVC to be aligned and sized more easily.

Interaction PVCs should allow further Interaction PVCs to be attached to them, making use of their own functionality. This would enable complex Interaction PVCs to be built from simple Interaction PVC components.

Ultimately, a Tool-set is also required which assists the interface designer in implementing new Interaction PVCs. This Tool-set could generate the Smalltalk Class 'stubs', which can then be refined by the interface designer. The interface to such a Tool-set could itself be defined using PVC concepts.

Further work is required developing the concept of Linkage Slots. Linkage Slots should be allowed to contain actual Smalltalk code, as well as simple Smalltalk messages. As discussed in section 8.2.2.3, this would allow more knowledge to be contained within the Separation Controller. As a result the UIMS would be more flexible and better suited to solving interface problems of greater complexity.

Work is required in improving the existing Smalltalk window handling mechanism. A user should be allowed to open more than one Part PVC direct manipulation interface at once. Information can then be moved between different Part PVC windows. A window is opened as the result of executing Smalltalk code. If this code appears as part of a group of Smalltalk statements, then further code must be executed when the opened window is later closed. In the case of Part PVC interface windows, the result of closing a

window is to return the Part PVCs attached object. This object may then be used by the Smalltalk code in which the opening statement is embedded.

Smalltalk supports multi-windowing. However, once a window is opened, the code which instantiated the opening is terminated and no value returned upon closing. If this were not the case, then problems could arise from closing the parent window before the child window. In Smalltalk this would leave the child window looking to complete the Smalltalk code which instantiated it. If the parent is already closed, then this code would not be found, and the system would be left 'hanging'. Instead, closing a parent window should automatically close all of the spawned children, and grand-children windows. This requires modification to the Smalltalk window handler.

Difficulties with PVC window sizing need addressing. If a PVC virtual window is mapped onto too small a physical window, problems arise. At best, the display cannot be understood because it is too cramped, and at worst, errors may result from within the PVC presentation code. PVCs should be allowed to specify their minimum physical window size. The interface designer can then be forced to set the physical window size accordingly. Alternatively, the View presentation functions can be ignored if the physical window size is too small. Then, when the mouse pointer points at a particular PVC, an exploded view may be displayed. This same view can then be use for capturing user interaction.

Scrolling is another area for further work. This would enable partial PVC Views to be displayed. Scrolling would be required in both the vertical and horizontal planes.

9.2.2. Quantitative User Model.

Further work is needed to extend the Quantitative User Model presented in chapter five. The heuristics, knowledge base, and implementation could all be improved.

A greater number of error types could be monitored according to a specific error classification. Further Learning Curves for individual classes and messages could be added. The relationships between different classes, messages, and applications also needs to be described. This would allow the

side effects of learning from different message, class, and application usage to be inferred.

A great deal of empirical work is required to determine the various heuristics and knowledge relationships described. This work is necessary if the Quantitative User Model is to truly represent the real user. Further knowledge can also be added, with a history of user interactions being maintained between different sessions. Knowledge is also required about the background of a user with other systems, and their personal preferences.

The Quantitative User Model should be implemented as an intrinsic part of Smalltalk, rather than as a simulation of user interactions. As the Smalltalk application is used, the message interpreter should automatically update the Quantitative User Model.

9.2.3. Part Hierarchies.

Work is required to separate the Part Hierarchy implementation and the UIMS. The Part Hierarchy should again be implemented as an intrinsic component of Smalltalk. Meanwhile, the UIMS architecture can also be implemented as a separate intrinsic component. Presently the two mechanisms are implemented together. This has the effect of slowing down the working of the UIMS. This is misleading, as the UIMS is itself fast.

Further work is also required in examining other object relationships beside part aggregation. Various types of association relationships exist between real world objects, and these need to be explicitly represented within object oriented systems. For example, the relationships that exist between family members, the relationship between flight bookings and actual plane seats, and the relationship between students and their courses.

9.3. Further User Interface Management System Implementation.

The other Intelligent Interface modules presented in chapter three also need to be implemented if a complete UIMS for generating Intelligent Interfaces is to be realised. Abstract Classes can be used to implement the various modules. Communication between modules can take the form of Smalltalk messages, and a special Controller Class instance could coordinate their associations. Such a UIMS would be restricted to object oriented applications,

although it may be possible to provide links to other languages and environments.

Prolog provides a useful language for describing the knowledge required by the various expert system modules. Prolog is already successfully incorporated into the Smalltalk 80 system [SmalltalkV:1980]. This allows Prolog knowledge structures to be built and queried using Prolog-like statements. Further work is required in evaluating the potential of this Prolog sub-system for building knowledge bases within Smalltalk.

Individual models could be implemented as instances of the same Smalltalk Class. A general purpose Model Class could be implemented, and Sub-classes may implement their own specialist functions.

9.4. Smalltalk 80 Programming Language Extensions.

Further work is required in extending the Smalltalk 80 system itself. These extensions should help maximise the potential of Smalltalk as an object oriented system for designing usable, separable applications and interfaces.

Many applications are implemented within the same Smalltalk environment. These different applications may share the same Classes, but are rarely used simultaneously. This is due to the environment provided by Smalltalk. Rather than writing and compiling separate application programs, applications are implemented by extending the existing environment. Tools are required to isolate individual applications, and aid the designer in their job. These tools could be linked to some object oriented design method, and allow individual applications to be viewed, implemented, and modified separately. The relationships between communicating objects could be shown graphically, and specialist Browser and editor tools provided.

Smalltalk is often used as a prototyping tool. After prototyping is complete, the new system is usually re-coded in a conventional, compilable language. Smalltalk does not allow applications to be compiled into executable machine code. In order to run a particular application, the Smalltalk environment and interpreter must be loaded. This is a major weakness, and will probably prevent Smalltalk from being substantially used in the commercial sector. Work is therefore required to develop a Smalltalk compiler which can generate machine executable code.

The inherent Smalltalk Application Model could be utilised as the basis for formal documentation. As individual applications, classes, and methods are implemented, the programmer can be prompted to enter the appropriate user documentation. This documentation can then be used to generate on line help for the user. Similarly, users of the application should be able to add their personalized comments.

Although the implicit Smalltalk 80 Application Model is expressive, further development is required. Application usage is usually goal directed. The user typically has a goal they wish to achieve, and performs a sequence of tasks in order to accomplish it. These tasks represent further sub-goals and often comprise of smaller tasks. Smalltalk must be able to represent this goal mechanism. A means of relating individual objects to a task / goal model is required. The methods which an object can understand must be related to tasks which they fulfil. Such an extension would enable intelligent goal inference mechanisms to be used to predict the intentions of a user, suggest alternative and more efficient task sequences, and intelligently correct any errors which arise.

9.5. Further Systems Analysis and Design Work.

The final interface, the implementation language and Tool-set, and the Systems Analysis and Design process are closely related. Further work is required to develop complete integrated implementation and design approaches, along with any necessary tools.

Complete Systems Analysis and Design methodologies are required which take full advantage of new UIMS. These must encompass both the information system requirements of an organisation, and the user interface requirements of its personnel. Work is also needed to empirically test and prove these methodologies.

New user centred methods are very much a long term objective. In the short term, work is required in drawing attention to the importance of the user in the Systems Analysis and Design processes. Systems analysts and designers need to be made aware of the importance of building computer systems that not only meet an organisations information system requirements, but which also meet the needs of its eventual users. The various factors which affect user acceptance of a computer system need highlighting, and Systems

Analysts and Designers must be taught how to utilise these factors in favour of the user.

Final Conclusions.

The research described has established that the acceptance of computer software by the user is becoming an important goal within the field of computing science. This goal is affected by a wide range of complex interrelated factors, which can be grouped according to actual computer software and hardware influences, the effects of the Systems Analysis and Design process, or the effect of the personal work environment of the user. The field of Human Computer Interaction research has arisen out of the need to address these factors and influences.

The research has successfully accomplished the objectives outlined in chapter one. In doing so, it has added to the body of knowledge within the field of Human Computer Interaction. The main contribution is concerned with the development of separable interface software architectures, and the identification of the potential benefits and constraints which result from their usage. Other contributions include the development of an object oriented User Interface Management System and integrated interface design Tool-set, an object oriented Part Hierarchy implementation, a practical investigation into Quantitative User Modelling, and a general overview of the diverse Human Computer Interaction field.

The major software design features which influence user acceptance of computer systems were identified as Simplicity, Consistency, Integration and Modes, Metaphor, the impact of various interaction styles such as Command and Natural Language, Menus, and Direct Manipulation, the use of Explicit Dialogue Control Specifications and standardised Style Guides, Error Handling, Documentation and Tutorial Support, and Interface Ergonomics. A need for software features which can be adjusted to satisfy the requirements of individual users was also established as necessary, in order to deal with the variability of users, in terms of their experience and background.

The field of Artificial Intelligence was highlighted as a major contributor to solving the problems arising from satisfying the needs of individual users. Existing Artificial Intelligence theories and practice can be used to automatically select appropriate software features for individual users, and some existing techniques were described. The main application areas of

Artificial Intelligence were categorised as Intelligent Help Systems, User and Application Modelling, Adaptive Interfaces, and Intelligent Planning Aids.

The controlled experimental evaluation of computer software was examined as a means of developing new interface designs, and for identifying software features which improve the 'usability' of computer systems. The importance and complexity of empirical evaluation was highlighted, using personal experience with the evaluation of a large library database user group. The related role of cognitive psychology research was also examined, and mechanisms for incorporating its research results into interface design suggested.

The need for improved Systems Analysis and Design methods was discussed, with emphasis placed upon designing systems for the user. The necessity of analysis techniques for eliciting the personal requirements of individual users, and the need for design methods and support tools for incorporating these requirements were established. The Prototyping approach was shown to be better suited to the design of interactive software than the traditional system life cycle method. Systems Analysis and Design methods were also proven to be dependent upon the availability of interface design and implementation tools. The influence of separation, and interface design tools incorporating separation, upon Systems Analysis and Design methods were discussed. The major influence was identified as the possible dichotomy in Systems Analysis and Design roles, resulting in separate interface design and application design disciplines.

Existing approaches to interactive software design were examined, and the need for improved software architectures identified. The use of new User Interface Management Systems was focussed upon, and a model for future interface software architectures suggested. The potential of User Interface Management Systems was successfully demonstrated with the implementation of a new object oriented User Interface Management System. This was implemented in Smalltalk 80, and was based upon and tested the concept of three component interface separation. The proposed architecture also showed the benefit of integrated interface design tools as a means of interactively designing consistent, 'usable', user interfaces.

The research demonstrated that the effects of software upon user acceptance could be best addressed if a line is drawn between the application software, and user interface software. It was shown that distinct separation of the user

interface and application is only possible with the introduction of a third separation component. Arguments based upon practical experience and the evaluation of alternative approaches to separation, established that two component separation is insufficient to meet the flexible requirements of new User Interface Management Systems. The many advantages arising from the use of three component separation were also discussed, and the constraints of proper separation detailed.

The advantages of the application of object oriented concepts to software design were discussed. The major advantage was identified as the similarity between the underlying object oriented paradigm and that of the real world. This facilitates the implementation of applications which can be easily understood by non expert users. The explicit support of separation within an object oriented language was also shown to assist the implementation of new User Interface Management Systems, based upon interface separation.

Results from this investigation should have an application in both the commercial and research sectors. As detailed in chapter nine, further refinements to the proposed User Interface Management System ought to provide a professional object oriented User Interface Management System in Smalltalk 80. Such a system would be of benefit to commerce, where Smalltalk 80 is already used as a Prototyping tool for interactive software. Future research in the field of Human Computer Interaction should also benefit from the proposed separation techniques. Other elements of the research may also provide the basis for further Human Computer Interaction investigations.

Bibliography

[Abrams, K.H:1987]

"Who's the Boss ?: Talking to Your Computer in the Artificial Intelligence Age."

Kenneth H. Abrams.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Adhami, E:1986]

"Emancipating the User Interface from Application System Semantics: An Application Expert."

E. Adhami, S.K. Mitra, D.P. Browne.

Internal Paper: British Telecom Research Laboratories. 1986.

[Adhami, E:1987]

"Application Modelling for the Provision of an Adaptive User Interface: A Knowledge Based Approach."

E. Adhami, D. P. Browne, S. K. Mitra.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Alexander, H:1986]

"Formally Based Techniques for Designing Human Computer Dialogues."

H. Alexander.

Stirling University, Computer Science Dept. September 1986, Number: RGP 35

[Alexander, H:1987]

"Executable Specifications as an Aid to Dialogue Design."

Heather Alexander.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Alty, J.L:1987]

"The Role of the Dialogue System in a User Interface Management System."

J. L. Alty, J. Mullin.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Andriole, S.J:1986]

"Graphic Equivalence, Graphic Explanations, and Embedded Process Modelling for Enhanced User-System Interaction."

Stephen J. Andriole.

IEEE Transactions on Systems, Man, and Cybernetics. 1986, Volume: 16, Number: 6.

[Andriole, S.J:1986b]

"Intelligent Aids for Tactical Planning."

Stephen J. Andriole, Harlan H. Black, Gerald W. Hopple, John R. Thompson.

IEEE Transactions on Systems, Man, and Cybernetics. 1986, Volume: 16, Number: 6.

[Apperley, M.D:1989]

"Lean Cuisine: A Low Fat Notation for Menus."

M. D. Apperley, R. Spence.

Interacting With Computers. 1989, Volume: 1, Number: 1.

[Bailey, P:1986]

"Speech Communication."

Peter Bailey.

Book: The User Interface: Human Factors in Computer Based Systems. Publishers - York University. 1986.

[Bailin, S.C:1989]

"An Object Oriented Requirements Specification Method."

Sidney C. Bailin.

Communications of the ACM. May 1989, Volume: 32, Number: 5.

[Balzert, H:1987]

"Objectives for the Humanisation of Software: A New and Extensive Approach."

Helmut Balzert.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Banerjee, J:1987]

"Data Model Issues for Object Oriented Applications."

Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, Won Kim, Darrel Woelk,
Nat Ballou.

ACM Transactions on Office Information Systems. January 1987,
Volume: 5, Number: 1.

[Barlow, J:1989]

"Interacting WITH Computers."

Judith Barlow, Roy Rada, Dan Diaper.

Interacting With Computers. 1989, Volume: 1, Number: 1.

[Barnard, P:1986]

"Human Computer Dialogues with Interactive Systems."

Phil Barnard, Nick Hammond.

Book: The User Interface: Human Factors in Computer Based Systems.

Publishers - York University. 1986.

[Barnard, P:1988]

"Approximate Modelling of Cognitive Activity with an Expert System: A
Theory Based Strategy for Developing an Interactive Design Tool."

Phil Barnard, M. Wilson, A. Maclean.

The Computer Journal. 1988, Volume: 31, Number: 5.

[Barnes, J.G.P:1980]

"An Overview of Ada."

J. G. P. Barnes.

Software - Practise and Experience. 1980, Number: 10

[Barstow, D.R:1984]

"Interactive Programming Environments."

David R. Barstow, Howard E. Shrobe, Eric Sandewall.

Publishers - McGraw-Hill Book Co, London, U.K. 1984.

[Begeman, M.L:1988]

"The Right Tool for the Job."

Michael L. Begeman, Jeff Conklin.

Byte Magazine. October 1988.

[Bell, D:1987]

"Software Engineering: A Programming Approach."

Doug Bell, Ian Morrey, John Pugh.

[Benbasat, I:1984]

"Command Abbreviation Behaviour in Human Computer Interaction."

Izak Benbasat, Yair Wand.

Communications of the ACM. April 1984, Volume: 27, Number: 4.

[Benbasat, I:1986]

"An Experimental Program Investigating Color Enhanced and Graphical Information Presentation: An Integration of the findings."

Izak Benbasat, Albert S. Dexter, Peter Todd.

Communications of the ACM. November 1986, Volume: 29, Number: 11.

[Bench-Capon, T.J.M:1989]

"People Interact through Computers, not with them."

T. J. M. Bench-Capon, A. M. McEnry.

Interacting With Computers. 1989, Volume: 1, Number: 1.

[Benest, I.D:1987]

"A Humanised Interface to an Electronic Library."

I. D. Benest, G. Morgan, M. D. Smithurst.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Bennett, J.L:1987]

"Developing a User Interface Technology for use in Industry."

John L. Bennett, Douglas J. Lorch.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Benyon, D:1987]

"System Adaptivity and the Modelling of Stereotypes."

David Benyon, Peter Innocent, Dianne M. Murray.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Benyon, D:1988]

"Modelling Users Cognitive Abilities in an Adaptive System."

David Benyon, Diane Murray, Steve Milan.

Internal Paper: Faculty of Mathematics, Open University, UK. 1988.

[Benyon, D:1988b]

"Experience with Adaptive Interfaces."

David Benyon, Diane Murray.

The Computer Journal. 1988, Volume: 31, Number: 5.

[Benzon, B:1985]

"The Visual Mind and The Macintosh."

Bill Benzon.

Byte Magazine. January 1985.

[Bez, H.E:1987]

"A Formal Design Methodology for End-User Interfaces: A Small Case Study Based on UNICON."

H. E. Bez, D. J. Cooke.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Birtwistle, G.M:1973]

"Simula begin."

Graham M. Birtwistle, O. J. Dahl, Bjorn Myhrhaug, Kristen Nygaard.

Publishers - Auerbach Publishers Inc, Philadelphia, U.S.A. 1973.

[Bjorn-Andersen, N:1988]

"Are 'Human Factors' Human ?"

N. Bjorn-Andersen.

The Computer Journal. 1988, Volume: 31, Number: 5.

[Blake, E:1987]

"On Including Part Hierarchies in Object Oriented Languages, with an Implementation in Smalltalk."

Edwin Blake, Steve Cook.

Proceedings for the 1988 ECOOP Conference. Publishers - Springer-Verlag, New York, N.Y., U.S.A. 1987.

[Boehm, B.W:1987]

"Improving Software Productivity."

Barry W. Boehm.

IEEE Computer. September 1987.

[Booch, G:1986]

"Object Oriented Development."

Grady Booch.

IEEE Transactions on Software Engineering. February 1986, Volume: 12,
Number: 2.

[Booth, P:1987]

"An Evaluative Classification of Mismatch Between Human and
Computer (ECM)."

Paul Booth.

Internal Paper: The HCI Research Unit, Huddersfield Polytechnic, UK.
July 1987.

[Borning, A:1987]

"Deltatalk: An Empirically and Aesthetically Motivated Simplification of
the Smalltalk 80 Language."

Alan Borning, Tim O'Shea.

Proceedings for the 1988 ECOOP Conference. Publishers - Springer-
Verlag, New York, N.Y., U.S.A. 1987.

[Borning, A.H:1982]

"Multiple Inheritance in Smalltalk-80."

Alan H. Borning, Daniel H. H. Ingalls.

Proceedings for the 1982 American Association for Artificial Intelligence
National Conference, Pittsburgh. Publishers - Morgan Kaufmann, Los
Altos, California, USA. 1982.

[Breuker, J:1988]

"Coaching in Help Systems."

Joost Breuker.

Book: Artificial Intelligence and Human Learning: Intelligent Computer
Aided Instruction. Publishers - Chapman and Hall, London, UK. 1987.

[Bright, P:1988]

"Digital Research's Graphics Environment Manager (GEM)."

Peter Bright.

Personal Computer World Magazine. March 1988.

[Brown, J.W:1982]

"Controlling the Complexity of Menu Networks."

James W. Brow.

Communications of the ACM. July 1982, Volume: 25, Number: 7.

[Brown, M.J:1983]

"The Complete Information Management System."

Michael J. Brown.

Byte Magazine. December 1983.

[Browne, D.P:1986]

"The Formal Specification of Adaptive User Interfaces using Command Language Grammar."

Dermot P. Browne, Brian D. Sharrat, Michael A. Norman.

Proceedings for the 1986 CHI Conference. Publishers - OO. April 1986.

[Browne, D.P:1987]

"Metrics for the Building, Evaluation, and Comprehension of Self Regulating Adaptive Systems."

Dermot P. Browne, Robert Trevellyen, Peter Totterdell, Mike Norman.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Budde, R:1984]

"Approaches to Prototyping: A Collection of Papers."

R. Budde (editor).

Publishers - Springer-Verlag, New York, N.Y., U.S.A. 1984.

[Bullinger, H.J:1987]

"Technology Assessment Concerning Impacts of Information Systems."

Hans-Jorg Bullinger, Klaus Kornwachs.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Burroughs:1983]

"An Ergonomic Overview."

Internal Paper: Burroughs Machines Limited, UK. April 1983.

[Byte:1981]

"Special Edition on Object Oriented Programming."

Byte Magazine. August 1981.

[Cameron, J.R:1986]

"An Overview of Jacksons Structured Design."

John R. Cameron.

IEEE Transactions on Software Engineering. February 1986, Volume: 12,
Number: 2.

[Campbell, F.W:1974]

"Contrast and Spacial Frequency."

Fergus W. Campbell, Lamberto Maffei.

Scientific American. November 1974, Volume: 231, Number: 5.

[Canter, D:1985]

"Characterizing User Navigation through Complex Data Structures."

David Canter, Rod Rivers, Graham Storrs.

Behaviour and Information Technology. 1985, Volume: 4, Number: 2.

[Carberry, S:1988]

"Modelling the User's Plans and Goals."

Sandra Carberry.

Computational Linguistics. September 1988, Volume: 14, Number: 3.

[Card, S.K:1983]

"The Psychology of Human Computer Interaction."

S. K. Card, T. P. Moran, A. Newell.

Publishers - Erlbaum, Hillsdale, New Jersey, U.S.A. 1983.

[Card, S.K:1987]

"CATALOGUES: A Metaphor for Computer Application Delivery."

Stephen K. Card, D. Austin Henderson Jnr.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers -
Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Carroll, J.M:1983]

"Presentation and Form in User Interface Architecture."

John M. Carroll.

Byte Magazine. December 1983.

[Carroll, J.M:1986]

"LisaLearning."

John M. Carroll, Sandra A. Mazur.

[Carroll, J.M:1987]

"Interface Design Issues for Advice Giving Expert Systems."

John M. Carroll, Jean McKendree.

Communications of the ACM. January 1987, Volume: 30, Number: 1.

[Carroll, J.M:1988]

"Learning by Doing with Simulated Intelligent Help."

John M. Carroll, Amy P. Aaronson.

Communications of the ACM. September 1988, Volume: 31, Number: 9.

[Carter, J.A:1987]

"The Basis for User-Oriented Context Sensitive Functions."

James A. Carter Jnr.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Cawsey, A:1989]

"Explanatory Dialogues."

Alison Cawsey.

Interacting With Computers. 1989, Volume: 1, Number: 1.

[Chang, D:1983]

"An Introduction to Integrated Software."

Dash Chang.

Byte Magazine. December 1983.

[Chubb, G.P:1981]

"SAINT, A Digital Simulation Language for the Study of Manned Systems."

G. P. Chubb.

Book: Manned System Design: Methods, Equipment, and Applications. Publishers - Plenum, New York, N.Y., U.S.A. 1981.

[Clark, I.A:1987]

"Designing a User Interface by Minimizing Cognitive Complexity."

Ian A. Clark.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Clarke, A.A:1986]

"A Three Level Human Computer Interface Model."

A. A. Clarke.

International Journal Man-Machine Studies. 1986, Number: 24

[Clowes, I:1985]

"User Modelling Techniques for Interactive Systems."

I. Clowes, I. Cole, F. Arshad, C. Hopkins, A. Hockley.

Proceedings for the 1985 HCI Conference. Publishers - Cambridge Press.
1985.

[Cockton, G:1986]

"Where do we Draw the Line ? - Derivation and Evaluatiuon of User Interface Software Separation Rules."

Gilbert Cockton.

Book: People and Computers: Designing for Usability. Publishers -
Cambridge Press. 1986.

[Cockton, G:1987]

"Interaction Ergonomics, Control and Separation: Open Problems in User Interface Management."

Gilbert Cockton.

Information and Software Technology. May 1987, Volume: 29, Number:
4.

[Cooper, M:1988]

"Interfaces that Adapt to the User."

Martin Cooper.

Book: Artificial Intelligence and Human Learning: Intelligent Computer
Aided Instruction. Publishers - Chapman and Hall, London, UK. 1988.

[Corbett, M:1987]

"Computerizing Data Presentation and Analysis."

M. Corbett, J. Kirakowski.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers -
Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Coutaz, J:1987]

"The Construction of User Interfaces and the Object Paradigm."

Joelle Coutaz.

Proceedings for the 1988 ECOOP Conference. Publishers - Springer-Verlag, New York, N.Y., U.S.A. 1987.

[CPlusPlus:1987]

"C++: Exposition and Experience."

Discussion.

Object Oriented Programming Society Newsletter. October 1987.

[Croft, W.B:1984]

"The Role of Context and Adaptation in User Interfaces."

Bruce W. Croft.

International Journal Man-Machine Studies. 1984, Number: 21

[Cutts, G:1987]

"Structured Systems Analysis and Design Methodology."

Geoff Cutts.

Publishers - Paradigm Publishing Ltd, London, U.K.. 1987.

[D'Arcy, B.G:1985]

"Development of an Issue Centred Methodology for Systems Investigation and Analysis."

Brian G. D'Arcy.

Internal Paper: Sheffield City Polytechnic, School of Computing and Management Science. 1985.

[Dean, M:1983]

"Simplify, Simplify, Simplify."

Martin Dean.

Byte Magazine. December 1983.

[Dearnley, P.A:1983]

"In Favour of System Prototypes and Their Integration into the Systems Development Cycle."

P. A. Dearnley, P. J. Mayhew.

The Computer Journal. 1983, Volume: 26, Number: 1.

[DeMichiel, L.G:1987]

"The Common Lisp Object Oriented System: An Overview."

Linda G. DeMichiel, Richard P. Gabriel.

Proceedings for the 1988 ECOOP Conference. Publishers - Springer-Verlag, New York, N.Y., U.S.A. 1987.

[Desmarais, M.C:1987]

"The Diagnosis of User Strategies."

Michel C. Desmarais, Serge Larochelle, Luc Giroux.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Deutsch, L.P:1981]

"Building Control Structures in the Smalltalk 80 System."

Peter L. Deutsch.

Byte Magazine. August 1981.

[Diaper, D:1987]

"POMESS:A People Oriented Methodology for Expert System Specification."

Dan Diaper.

Alvey HI Club, July 1987. 1987.

[Dillon, A:1987]

"A Psychological View of User Friendliness."

Andrew Dillon.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Dillon, A:1988]

"Reading from Paper Versus Reading From Screen."

A. Dillon, C. McKnight, J. Richardson.

The Computer Journal. 1988, Volume: 31, Number: 5.

[Dirlich, G:1987]

"Integration at a Work Place for Statistical Consulting."

G. Dirlich, H. Federkiel, E. Hansert, A. Yassouridis.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Dix, A:1987]

"Giving Control Back to the User."

Alan Dix.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Dodani, M.H:1989]

"Separation of Powers."

Maresh H. Dodani, Charles E. Hughes, Michael J. Moshell.

Byte Magazine. March 1989.

[Downs, E:1987]

"SSADM: Application and Context."

E. Downs, P. Clare, I. Coe.

Publishers - Prentice Hall International, London, U.K. 1987.

[Drake, K:1985]

"The Application of Metaphor to Constrained User Interface Design."

Kieron Drake.

Internal Paper: Queen Mary College, IRL. June 1985, Number: 361

[Dray, S.M:1987]

"Getting the Baby into the Bathwater: Putting Organisational Planning into the Systems Design Process."

Susan M. Dray.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Ducournau, R:1987]

"On Some Algorithms for Multiple Inheritance in Object Oriented Programming."

R. Ducournau, M. Habib.

Proceedings for the 1988 ECOOP Conference. Publishers - Springer-Verlag, New York, N.Y., U.S.A. 1987.

[Dunlavy, N:1986]

"The Use of Object Oriented Techniques for Programming the User Interface."

Nicholas Dunlavy.

Internal Paper: Queen Mary College, IRL. October 1986.

[Durham, T:1988]

"Hit or Myth ? A Hunt for an Elusive Beast (Adaptive Intelligent Dialogues)."

Tony Durham.

Computing. October 1988.

[Eason, K.D:1987]

"A User Centred Approach to the Design of a Knowledge Based System."

Ken D. Eason, S. D. P. Harker, P. F. Raven, J. R. Brailsford, A. D. Cross.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Eason, K.D:1988]

"The Supplier's Role in the Design of Products for Organisations."

Ken D. Eason, Susan Harker.

The Computer Journal. 1988, Volume: 31, Number: 5.

[Edmonds, E:1987]

"Good Software Design: What does it mean ?"

Ernest Edmonds.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Edwards, S:1983]

"Why is Software so Hard to Use."

Sam Edwards.

Byte Magazine. December 1983.

[Ege, R.K:1987]

"The Filter Browser. Defining Interfaces Graphically."

Raimund K. Ege, David Maier, Alan Borning.

Proceedings for the 1988 ECOOP Conference. Publishers - Springer-Verlag, New York, N.Y., U.S.A. 1987.

[Eklundh, K.S:1987]

"Digressional vs Semantic Subordination: On the Role of Menu Structures for User's Understanding of a Human Computer Dialogue."

Kerstin Severinson Eklundh.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Elkerton, J:1987]

"A Summary of Experimental Research on Command Selection Aids."

Jay Elkerton, Robert C. Williges.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Elsom-Cook, M:1988]

"Guided Discovery Tutoring and Bounded User Modelling."

Mark Elsom-Cook.

Book: Artificial Intelligence and Human Learning: Intelligent Computer Aided Instruction. Publishers - Chapman and Hall, London, UK. 1988.

[Erlandsen, J:1987]

"Intelligent Help Systems."

Jens Erlandsen, Jan Holm.

Information and Software Technology. April 1987, Volume: 29, Number: 3.

[Fountain, A.J:1985]

"Modelling User Behaviour with Formal Grammar."

A. J. Fountain, M. A. Norman.

Book: People and Computers: Designing the Interface. Publishers - Cambridge Press. 1985.

[Fowler, C.J.H:1987]

"Gender and Cognitive Style Differences at the Human Computer Interface."

C. J. H. Fowler, D. Murray.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Frohlich, D.M:1985]

"Requirements for an Intelligent Form Filling Interface."

D. M. Frohlich, L. P. Crossfield, G. N. Gilbert.

Book: People and Computers: Designing the Interface. Publishers - Cambridge Press. 1985.

[Galer, M:1987]

"The Presentation of Human Factors to Designers of I.T. Products."

Margaret Galer, A. J. Russell.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Garvey, M.A:1989]

"Introduction to Object Oriented Databases."

M. A. Garvey, M. S. Jackson.

Information and Software Technology. December 1989, Volume: 31, Number: 10.

[Gilbert, G.N:1987]

"Cognitive and Social Models of the User."

G. Nigel Gilbert.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Gimnich, R:1987]

"Constructive Formal Specifications for Rapid Prototyping."

Rainer Gimnich, Jurgen Ebert.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Glinert, E.P:1987]

"A (Formal) Model for (Iconic) Programming Environments."

Ephraim P. Glinert, Jakob Gonczarowski.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Godwin, A.N:1989]

"A Comparison of Jackson Structured Design and Data Flow Diagrams as Descriptive Tools."

A. N. Godwin, M. B. Gore, D. W. Salt.

The Computer Journal. 1989, Volume: 32, Number: 3.

[Goldberg, A:1981]

"Introducing the Smalltalk 80 System."

Adele Goldberg.

Byte Magazine. August 1981.

[Goldberg, A:1983]

"Smalltalk 80: The Language and its Implementation (The 'Blue Book')."

Adele Goldberg, David Robson.
Publishers - Addison-Wesley, U.K. 1983.

[Goldberg, A:1983b]

"Smalltalk 80: The Interactive Programming Environment (The 'Orange Book')."

Adele Goldberg.

Publishers - Addison-Wesley, U.K. 1983.

[Gould, J.D:1987]

"How to Design Usable Systems."

John D. Gould.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Grace, J.E:1987]

"The Man-Machine Interface: The Natural Language Barrier."

J. E. Grace.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Gray, P.M.D:1988]

"Expert Systems and Object Oriented Databases: Evolving a New Software Architecture."

Peter M. D. Gray.

Proceedings for the 1988 Expert Systems Conference. Publishers - Cambridge Press. 1988.

[Green, E:1990]

"Designing Systems, Defining Jobs: A Gender Perspective on the Development of Office Information Systems."

Eileen Green, Jenny Owen, Den Pain.

Internal Paper: Sheffield City Polytechnic, Department of Applied Social Studies. 1990.

[Green, T.R.G:1988]

"Formalisable Models of User Knowledge in Human Computer Interaction."

Thomas R. G. Green, Franz Schiele, Stephen J. Payne.

Book: Working With Computers: Theory versus Outcome. Publishers -
Van der Veer. 1988.

[Greenberg, S:1985]

"Adaptive Personalised Interfaces - A Question of Viability."

Saul Greenberg, Ian H. Witten.

Behaviour and Information Technology. 1985, Volume: 4, Number: 1.

[Grudin, J:1987]

"Social Evaluation of the User Interface: Who does the Work, and who gets the Benefit ?"

Jonathon Grudin.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers -
Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Hagelbarger, D.W:1983]

"Experiments in Teleterminal Design."

David W. Hagelbarger, Richard A. Thompson.

IEEE Spectrum. October 1983.

[Hagendorf, H:1987]

"A Framework of Developing Semantic Models of User Performance."

Herbert Hagendorf.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers -
Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Halbert, D.C:1987]

"User Types and Inheritance in Object Oriented Languages."

Daniel C. Halbert, Patrick D. O'Brien.

Proceedings for the 1988 ECOOP Conference. Publishers - Springer-
Verlag, New York, N.Y., U.S.A. 1987.

[Hanne, K.H:1987]

"Design and Implementation of Direct Manipulative and Deictic User Interfaces to Knowledge Based Systems."

K. H. Hanne, A. Grable.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers -
Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Harker, S:1988]

"The Use of Prototyping and Simulation in the Development of Large Scale Applications."

Susan Harker.

The Computer Journal. 1988, Volume: 31, Number: 5.

[Harland, D.M:1989]

"The REKURSIV and LINGO"

David M. Harland, Brian Drummond.

Internal Paper: Linn Smart Computing Limited, 257 Drakemire Drive,
Glasgow G45 9SN, Scotland, U.K. 1989.

[Harris, J.R:1987]

"Evaluation of Rapid Prototyping Methodology in a Human Interface."

J. R. Harris, D. W. Parker.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers -
Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Hartley, J.R:1988]

"Question Answering and Explanation Giving in On-Line Help Systems."

Roger J. Hartley, Michael J. Smith.

Book: Artificial Intelligence and Human Learning: Intelligent Computer
Aided Instruction. Publishers - Chapman and Hall, London, UK. 1988.

[Heckel, P:1983]

"Walt Disney and User Oriented Software."

Paul Heckel.

Byte Magazine. December 1983.

[Hecking, M:1987]

"How to Use Plan Recognition to Improve the Abilities of the Intelligent Help System SINIX Consultant."

Matthias Hecking.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers -
Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Hekmatpour, S:1987]

"Evolutionary Prototyping and the Human-Computer Interface."

S. Hekmatpour, D. C. Ince.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers -
Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Henskes, D.T:1987]

"Rapid Prototyping of Man-Machine Interfaces for Telecommunications Equipment Using Interactive Animated Computer Graphics."

D. T. Henskes, J. C. Tolmie.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Herbach, M:1983]

"The User Interface: Two Approaches."

Martin Herbach, Richard Katz, Joseph Landau.

Byte Magazine. December 1983.

[Hirschheim, R:1988]

"Information Systems and User Resistance: Theory and Practise."

R. Hirschheim, M. Newman.

The Computer Journal. 1988, Volume: 31, Number: 5.

[Hockley, A:1986]

"Adaptive Intelligent Dialogues."

Andrew Hockley.

Internal Paper: British Telecom Research Laboratories. 1986.

[Hood:1989]

"HOOD Manual, Issue 2.3, Draft A, February 1989."

CISI Ingenierie, Matra Espace.

ESTEC Repro Service February 1989.

[Hoppe, H.U:1985]

"A Survey of Models and Formal Description Methods in Human Computer Interaction with Example Applications."

H. U. Hoppe, M. Tauber, J. E. Ziegler.

ESPRIT Project 385 - Human Factors in Information Technology. 1985, Number: B.3.2a

[Horn, C:1987]

"Conformance, Genericity, Inheritance, and Enhancement."

Chris Horn.

Proceedings for the 1988 ECOOP Conference. Publishers - Springer-Verlag, New York, N.Y., U.S.A. 1987.

[Houston, T:1983]

"The Allegory of Software: Beyond, Behind, and Beneath the Electronic Desk."

Tom Houston.

Byte Magazine. December 1983.

[Hudson, S.E:1989]

"Cactis: A Self-Adaptive, Concurrent Implementation of an Object Oriented Database Managament System."

Scott E. Hudson, Roger King.

ACM Transactions on Database Systems. September 1989, Volume: 14, Number: 3.

[HUFIT:Overview]

"Human Factors in Information Technology: Project Overview."

Internal Paper: ESPRIT Project 385 - HUSAT Research Centre, Loughborough, UK. 1987.

[Hulme, C:1986]

"Language by Eye."

Charles Hulme.

Book: The User Interface: Human Factors in Computer Based Systems. Publishers - York University. 1986.

[Ingalls, D.H.H:1981]

"Design Principles Behind Smalltalk."

Daniel H. H. Ingalls.

Byte Magazine. August 1981.

[Ingalls, D.H.H:1981b]

"The Smalltalk Graphics Kernel."

Daniel H. H. Ingalls.

Byte Magazine. August 1981.

[Jagodzinski, A.P:1988]

"A Multidimensional Approach to the Measurement of Human Computer Performance."

A. P. Jagodzinski, D. D. Clarke.

[Jarke, M:1985]

"A Framework for Choosing a Database Query Language."

Matthias Jarke, Yannis Vassiliou.

Computing Surveys. September 1985, Volume: 17, Number: 5.

[Jeremaes, P:1987]

"Specifying a Logic of Dialogues."

P. Jeremaes.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). 1987.

[Jones, J:1988]

"Understanding User Behaviour in Command Driven Systems."

John Jones, Mark Millington, Peter Ross.

Book: Artificial Intelligence and Human Learning: Intelligent Computer Aided Instruction. Publishers - Chapman and Hall, London, UK. 1988.

[Kaehler, T:1981]

"Virtual Memory for an Object Oriented Language."

Ted Kaehler.

Byte Magazine. August 1981.

[Kantorowitz, E:1989]

"The Adaptable User Interface."

Eliezer Kantorowitz, Oded Sudarsky.

Communications of the ACM. November 1989, Volume: 32, Number: 11.

[Karat, J:1987]

"Evaluating User Interface Complexity."

John Karat, Richard Fowler, Mary Gravelle.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Kellog, W.A:1987]

"Conceptual Consistency in the User Interface."

Wendy A. Kellog.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Kemke, C:1987]

"Representation of Domain Knowledge in an Intelligent Help System."

Christel Kemke.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Kidd, A:1986]

"Expert Systems."

Alison Kidd.

Book: The User Interface: Human Factors in Computer Based Systems.

Publishers - York University. 1986.

[Kieras, D:1985]

"An Approach to the Formal Analysis of User Complexity."

D. Kieras, P. G. Polson.

International Journal Man-Machine Studies. 1985, Number: 22

[Kluger, L:1989]

"The Open Look Graphical User Interface and its Toolkits."

Larry Kluger.

IEEE Special Colloquium on "User Interface Management Systems".

November 1989.

[Kornwachs, K:1987]

"A Quantitative Measure for the Complexity of Man-Machine Interaction Process."

Klaus Kornwachs.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers -

Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Koved, L:1986]

"Embedded Menus: Selecting Items in Context."

Larry Koved, Ben Shneiderman.

Communications of the ACM. April 1986, Volume: 29, Number: 4.

[Kraak, J:1987]

"Multi-Level User Interfaces: Software Tools and an Application."

J. Kraak.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Kreutzer, W:1987]

"A Modeller's Workbench: Experiments in Object Oriented Simulation Programming."

Wolfgang Kreutzer.

Proceedings for the 1988 ECOOP Conference. Publishers - Springer-Verlag, New York, N.Y., U.S.A. 1987.

[Kristensen, B.B:1987]

"Classification of Actions or Inheritance also for Methods."

Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, Kristen Nygaard.

Proceedings for the 1988 ECOOP Conference. Publishers - Springer-Verlag, New York, N.Y., U.S.A. 1987.

[LaLonde, W:1989]

"Pluggable Tiling Windows"

Wilf LaLonde, John Pugh.

Journal of Object Oriented Programming. September 1989.

[Lambie, K:1989]

"Introducing the REKURSIV."

Kirstine Lambie.

Internal Paper: Linn Smart Computing Limited, 257 Drakemire Drive, Glasgow G45 9SN, Scotland, U.K. 1989.

[Lane, N.E:1981]

"The Human Operator Simulator: An Overview."

N. E. Lane, N. I. Strieb, F. A. Glenn, R. J. Wherry.

Book: Manned System Design: Methods, Equipment, and Applications. Publishers - Plenum, New York, N.Y., U.S.A. 1981.

[Lehner, P.E:1986]

"On the Role of Artificial Intelligence in Command and Control."

Paul E. Lehner.

IEEE Transactions on Systems, Man, and Cybernetics. 1986, Volume: 16, Number: 6.

[Lesniewski, A:1987]

"Designing a User-Oriented Interface to a Document Management System."

A. Lesniewski, H. Rossler, P. Szabo, K. H. Jerke.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Lindsjorn, Y:1988]

"Database Concepts Discussed in an Object Oriented Perspective."

Yngve Lindsjorn, Dag Sjoberg.

Proceedings for the 1988 ECOOP Conference. Publishers - Springer-Verlag, New York, N.Y., U.S.A. 1988.

[Loomis, M.E.S:1987]

"An Object Modelling Technique for Conceptual Design."

M. E. S. Loomis, A. V. Shah, J. E. Rumbaugh.

Proceedings for the 1988 ECOOP Conference. Publishers - Springer-Verlag, New York, N.Y., U.S.A. 1987.

[Lutze, R:1987]

"Customizing Help Systems to Task Structures and User Needs."

Rainer Lutze.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Maruichi, T:1987]

"Behavioural Simulation Based on Knowledge Objects."

Takeo Maruichi, Tetsuya Uchiki, Mario Tokoro.

Proceedings for the 1988 ECOOP Conference. Publishers - Springer-Verlag, New York, N.Y., U.S.A. 1987.

[Matilla, M:1987]

"Computer Aided Ergonomics Design: A Program for Suitable Control Locations."

Markku Matilla, Markku Leppanen.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[McCoy, K.F:1988]

"Reasoning on a Highlighted User Model to Respond to Misconceptions."

[Mevel, A:1987]

"Smalltalk-80."

A. Mevel, T. Gueguen.

Publishers - MacMillan Education Ltd, London, UK. 1987.

[Meyer, B:1985]

"On Formalism in Specifications."

Bertrand Meyer.

IEEE Software. January 1985.

[Meyer, B:1987]

"EIFFEL: Programming for Reusability and Extendibility."

Bertrand Meyer.

Object Oriented Programming Society, Newsletter. October 1987.

[Meyer, B:1987b]

"Object-Oriented Software Construction."

Bertrand Meyer.

Publishers - Prentice Hall International, London, U.K. October 1987.

[Miller, J.R:1987]

"The Role of the System Image in Intelligent User Assistance."

James R. Miller, William C. Hill, Jean Mckendree, Michael E. J. Masson,
Brad Blumenthal.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers -
Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Minkowitz, C:1987]

"A Formal Description of Object Oriented Programming Using Vienna
Development Method."

Cydney Minkowitz, Peter Henderson.

Internal Paper: University of Stirling, Stirling, FK9 4LA, Scotland. 1987.
Number: FPN-13

[Minor, S:1987]

"Structured Command Interaction Based on a Grammar Interpreting
Synthesizer."

Sten Minor.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Minstrel:1987]

"Project Minstrel - Technical Overview."

ESPRIT Project Number 59. 1987, Number: 156/1987-11-17/DDC36

[Moll, T:1987]

"Do People Really User On-line Assistance ?"

Thomas Moll, Roland Sauter.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Monk, A:1986]

"Artificial Intelligence."

Andrew Monk.

Book: The User Interface: Human Factors in Computer Based Systems. Publishers - York University. 1986.

[Monk, A:1986b]

"Introduction - Man as a Processor of Information."

Andrew Monk.

Book: The User Interface: Human Factors in Computer Based Systems. Publishers - York University. 1986.

[Monk, A:1986c]

"Principles of Experimental Design."

Andrew Monk.

Book: The User Interface: Human Factors in Computer Based Systems. Publishers - York University. 1986.

[Monk, A:1986d]

"Statistical Evaluation."

Andrew Monk.

Book: The User Interface: Human Factors in Computer Based Systems. Publishers - York University. 1986.

[Moran, T.P:1981]

"The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems."

Thomas P. Moran.

International Journal Man-Machine Studies. 1981, Number: 15

[Morris, D:1988]

"Human Computer Interface Recording."

D. Morris, C. J. Theaker, R. Phillips, W. Love.

The Computer Journal. 1988, Volume: 31, Number: 5.

[Multipoint:1989]

"Esprit Project - Multipoint Interactive Audiovisual Communication (MIAC)."

ESPRIT Project Number 1057. 1989.

[Multiworks:1989]

"Esprit Project - Multiworks."

ESPRIT Project Number 2105. 1989.

[Mumford, E:1979]

"A Participative Approach to Computer Systems Design: A Case Study of the Introduction of a New Computer System."

Enid Mumford, Don Henshall.

Publishers - Manchester Business School, Manchester, U.K. 1979.

[Mumford, E:1981]

"Values, Technology, and Work."

Enid Mumford.

Publishers - Nijhoff. 1981.

[Murray, D.M:1987]

"Embedded User Models."

Dianne M. Murray.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Nebeker, D.M:1987]

"Automated Monitoring, Feedback, and Rewards: Effects on Work-station Operator's Performance, Satisfaction, and Stress."

Delbert M. Nebeker.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Noah, W.W:1986]

"Adaptive User Interfaces for Planning and Decision Aids in C3I Systems."

William W. Noah, Stanley M. Halpin.

IEEE Transactions on Systems, Man, and Cybernetics. 1986, Volume: 16, Number: 6.

[Norcio, A.F:1989]

"Adaptive Human-Computer Interfaces: A Literature Survey and Perspective."

Anthony F. Norcio, Jaki Stanley.

IEEE Transactions on Systems, Man, and Cybernetics. March 1989, Volume: 19, Number: 2.

[Norman, D.A:1986]

"Cognitive Engineering: User Centred System Design."

D. A. Norman, S. W. Draper.

Publishers - Erlbaum, Hillsdale, New Jersey, U.S.A. 1986.

[Novara, F:1987]

"Usability Evaluation and Feedback to Designers: An Experimental Study."

F. Novara, N. Bertaggia, N. Allamanno.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[NTIS:1987]

"Design Guidelines for User System Interface Software."

Reproduced by MicroInfo Ltd.

MicroInfo Ltd, PO Box 3, Alton, Hampshire, GU34 2PG. 1986.

[Oddy, R.N:1977]

"Information Retrieval Through Man Machine Dialogue."

R. N. Oddy.

The Journal of Documentation. March 1977, Volume: 33, Number: 1.

[Ogden, W.C:1987]

"What do Users Say to Their Natural Language Interface ?"

William C. Ogden, Ann Sorknes.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Oldenburg, H:1989]

"OSF MOTIF: The User Interface Standard."

H. Oldenburg.

IEEE Special Colloquium on "User Interface Management Systems". November 1989.

[Ord, J.G:1989]

"Who's Joking ? The Information System at Play."

Jacqueline G. Ord.

Interacting With Computers. 1989, Volume: 1, Number: 1.

[Parkin, A:1980]

"Systems Analysis."

Andrew Parkin.

Publishers - Edward Arnold Ltd, London, U.K. 1980.

[Patel, H:1989]

"Open Dialogue"

Hitash Patel.

IEEE Special Colloquium on "User Interface Management Systems". November 1989.

[Paton, N.W:1989]

"A Rule Based Query Optimiser for Object Oriented Databases."

Norman W. Paton, Peter M. D. Gray.

Internal Paper: Department of Computing Science, University of Aberdeen, Scotland. 1989.

[Paul, D.W:1987]

"An Approach Towards a Truly High-Level and Integrated User-Computer Interface."

Dietrich W. Paul, Hans R. Wiehle.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Payne, S.J:1986]

"Task Action Grammars: A Model of the Mental Representation of Task Languages."

Stephen J. Payne, Thomas R. G. Green.

Human Computer Interaction 1986, Number: 2

[Payne, S.J:1987]

"Complex Problem Spaces: Modelling the Knowledge Needed to Use Interactive Devices."

Stephen J. Payne.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Pearce, D:1989]

"HyperNeWS: An Interactive Design Tool."

Danny Pearce.

IEEE Special Colloquium on "User Interface Management Systems". November 1989.

[Phillips, C:1986]

"Smalltalk V: Screentest."

Carl Phillips.

Personal Computer World Magazine. November 1986.

[Pollack, M:1986]

"A Model of Plan Inference that Distinguishes Between the Belief of Actors and Observers."

Martha Pollack.

Proceedings for the 1986 24th Annual Meeting of the Association for Computational Linguistics, New York, NY. 1986.

[Poole, F:1988]

"DB4GL: An Intelligent Database System."

Frank Poole, Bryn Hird.

Internal Paper: Sheffield City Polytechnic, School of Computing and Management Science. 1988.

[Pope, A:1983]

"Making Life Easier for Professional and Novice Programmers."

Andy Pope, Geoff Kates, Dan Fineberg.

[Poulson, D.F:1987]

"The Use of Participative Exercises in Human Factors for Education and Design."

D. F. Poulson, C. A. Johnson, J. Moulding.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Pountain, D:1988]

"REKURSIV: An Object Oriented CPU."

Dick Pountain.

Byte Magazine. November 1988.

[Pratt, J.M:1987]

"The Social Impact of User Models."

John M. Pratt.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Pullinger, D.J:1989]

"Moral Judgements in Designing Better Systems."

David J. Pullinger.

Interacting With Computers. 1989, Volume: 1, Number: 1.

[Quesne, P.N:1988]

"Individual and Organisational Factors and the Design of Integrated Program Support Environments."

P. N. Le Quesne.

The Computer Journal. 1988, Volume: 31, Number: 5.

[Quint, V:1987]

"An Abstract Model for Interactive Pictures."

V. Quint, I. Vatton.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Rasmussen, J:1987]

"Cognitive Engineering."

Jens Rasmussen.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Rawlings, R:1989]

"Objective C: An Object-Oriented Language for Pragmatists."

Rosamund Rawlings.

IEEE Special Colloquium on "Applications of Object Oriented Programming". November 1989.

[Reenskaug, T.M.H:1981]

"User Oriented Descriptions of Smalltalk Systems."

Trygve M. H. Reenskaug.

Byte Magazine. August 1981.

[Reid, P:1986]

"Workstation Design: Devices, Activities, and Display Techniques."

Pete Reid.

Book: The User Interface: Human Factors in Computer Based Systems. Publishers - York University. 1986.

[Riekert, W:1987]

"The ZOO Metasystem: A Direct Manipulation Interface to Object Oriented Knowledge Bases."

Wolf-Fritz Riekart.

Proceedings for the 1988 ECOOP Conference. Publishers - Springer-Verlag, New York, N.Y., U.S.A. 1987.

[Rivers, R:1989]

"Embedded User Models - Where Next ?"

Rod Rivers.

Interacting With Computers. 1989, Volume: 1, Number: 1.

[Robson, D:1981]

"Object Oriented Software Systems."

David Robson.

Byte Magazine. August 1981.

[Rogers, Y:1989]

"Icons at the Interface: Their Usefulness."

Yvonne Rogers.

[Rosenberg, J:1983]

"Evaluating the Suggestiveness of Command Names."

J. Rosenberg.

Proceedings for the 1983 ACM Human Factors in Computing Systems Conference. Publishers - Gaithersburg, MD, USA. March 1983.

[Rouse, W.B:1986]

"Understanding and Enhancing User Acceptance of Computer Technology."

William B. Rouse, Nancy M. Morriss.

IEEE Transactions on Systems, Man, and Cybernetics. 1986, Volume: 16, Number: 6.

[Samurcay, R:1987]

"Design Systems for Training and Decision Aids: Cognitive Task Analysis as a Prerequisite."

Renan Samurcay, Janine Rogalski.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Sandberg, J:1988]

"Research on HELP Systems: Emperical Study and model construction."

Jacobijn Sandberg , Joost Breuker, Radboud Winkels.

Proceedings for the 1988 European Conference on Artificial Intelligence, Munich. 1988.

[Schroder, M:1988]

"Evaluating User Utterances in Natural Language Interfaces to Databases."

Martin Schroder.

Computers and Artificial Intelligence. 1988, Volume: 7, Number: 4.

[Seidewitz, E:1986]

"Towards a General Object Oriented Software Development Methodology."

Ed Seidewitz, Mike Stark.

Proceedings for the 1986 First International Conference on ADA Programming Language Applications. 1986.

[Self, J:1988]

"Artificial Intelligence and Human Learning: Intelligent Computer Aided Instruction."

J. Self (editor).

Publishers - Chapman and Hall, London, UK. 1988.

[Sharrat, B.D:1987]

"Top-down Interactive Systems Design: Some Lessons Learnt From Using Command Language Grammar."

B. D. Sharrat.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Shneiderman, B:1983]

"Direct Manipulation: A Step Beyond Programming Languages."

Ben Shneiderman.

IEEE Computer. August 1983.

[Shneiderman, B:1987]

"Designing the User Interface. Strategies for Effective Human Computer Interaction."

Ben Shneiderman.

Publishers - Addison-Wesley, U.K. 1987.

[Smalltalk80:ReferenceGuide]

"The Smalltalk-80 Programming System. Virtual Image Version VI 2.2. Reference Guide and Release Notes."

Publishers - Parc Place Systems, Palo Alto, California. 1988.

[Smalltalk80:1981]

"The Smalltalk 80 System."

Xerox Learning Research Group.

Byte Magazine. August 1981.

[SmalltalkV:1980]

"Smalltalk/V: Goodies Extension Pack."

Reference Manual.

Publishers - Digitalk Inc, Los Angeles, California, U.S.A. 1980.

[Smith, J.J:1985]

"SUSI - A Smart User-System Interface."

J. Jerrams Smith.

Book: People and Computers:Designing the Interface. Publishers -
Cambridge Press. 1985.

[Smith, S.L:1984]

"The User Interface to Computer Based Information Systems: A Survey of
Current Software Design Practise."

Sidney L. Smith, Jane N. Mosier.

Proceedings for the 1984 Interact Conference Proceedings. Publishers -
Elsevier Science Publishers B. V. (North-Holland). 1984.

[Sneeringer, J:1978]

"User-Interface Design for Text Editting: A Case Study."

J. Sneeringer.

Software - Practice and Experience 1978, Number: 8

[Snowberry, K:1985]

"Effect of Help Fields on Navigating Through Hierarchical Menu
Structures."

Kathleen Snowberry, Stanley Parkinson, Norwood Sisson.

International Journal Man-Machine Studies. 1985, Number: 22

[Somiw:1989]

"Esprit Project - Secure, Open, Multimedia, Integrated Workstation
(SOMIW)."

ESPRIT Project Number 367. 1989.

[Sommerville, I:1985]

"Software Engineering."

Ian Sommerville.

Publishers - Addison-Wesley, U.K. 1985.

[Spall, R.P:1986]

"A Program For Responding to Political Statements from Different
Ideological Points of View."

Roger P. Spall.

Thesis: BSc Dissertation, Sheffield City Polytechnic, School of Computing
and Management Science. March 1986.

[Spall, R.P:1988]

"A Generalised Human-Machine Interface for Proprietary Software."

Roger P. Spall.

Thesis: MPhil to PhD Transfer Report, Sheffield City Polytechnic, School of Computing and Management Science. June 1988.

[Spall, R.P:1990]

"An Investigation into the Quantitative User Modelling of User Interactions for the purpose of Predicting User Expertise."

R. P. Spall, R. A. Steele.

Proceedings for the 1990 Interact Conference (Cambridge). Publishers - Elsevier Science Publishers B. V. (North-Holland). 1990.

[Sproull, R.F:1983]

"Challenges in Graphical User Interfaces."

R. F. Sproull.

Proceedings for the 1983 Joint IBM/Newcastle University Seminar. Publishers - Computing Laboratory, University of Newcastle Upon Tyne. 1983.

[Stroustrup, B:1986]

"The C++ Programming Language."

Bjarne Stroustrup.

Publishers - Addison-Wesley, U.K. 1986.

[Stroustrup, B:1987]

"What is "Object Oriented Programming" ?"

Bjarne Stroustrup.

Proceedings for the 1988 ECOOP Conference. Publishers - Springer-Verlag, New York, N.Y., U.S.A. 1987.

[Sun:1990]

"Software Technical Bulletin: Window System Evolution."

Sun Microsystems, Technical Information Services. March 1990. Number: 1990-03

[Sutcliffe, A:1989]

"Task Analysis, Systems Analysis, and Design: Symbiosis or Synthesis ?"

Alistair Sutcliffe.

Interacting With Computers. 1989, Volume: 1, Number: 1.

[Sutcliffe, A.G:1987]

"Do Users Know they have User Models ? Some Experiences in the Practise of User Modelling."

A. G. Sutcliffe, A. C. Old.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Swinehart, D.C:1974]

"Copilot: A Multiple Process Approach to Interactive Programming Systems."

D. C. Swinehart.

Thesis: Stanford Artificial Intelligence Laboratory, Stanford University, USA. July 1974.

[Tauber, M.J:1988]

"On Mental Models and the User Interface."

Michael J. Tauber.

Book: Working with Computers: Theory versus Outcome. Publishers - Van der Veer. 1988.

[Tesler, L:1981]

"The Smalltalk Environment."

Larry Tesler.

Byte Magazine. August 1981.

[Thimbleby, H:1983]

"Dialogue Design: Principle or Prejudice? 'Generative User-Engineering Principles'."

Harold Thimbleby.

Internal Paper: University of London, UK. 1983.

[Thimbleby, H:1986]

"Basic User Engineering Principles for Display Editors."

Harold Thimbleby.

Book: The User Interface: Human Factors in Computer Based Systems. Publishers - York University. 1986.

[Thomas, D:1989]

"What's in an Object."

Dave Thomas.

Byte Magazine. March 1989.

[Thompson, P:1986]

"Visual Perception."

Peter Thompson.

Book: The User Interface: Human Factors in Computer Based Systems.
Publishers - York University. 1986.

[Thomson, N:1986]

"Human Memory."

Neil Thomson.

Book: The User Interface: Human Factors in Computer Based Systems.
Publishers - York University. 1986.

[Thomson, N:1986b]

"Thinking and Reasoning."

Neil Thomson.

Book: The User Interface: Human Factors in Computer Based Systems.
Publishers - York University. 1986.

[Totterdell, P.A:1986]

"Design and Evaluation of the Adaptive Intelligent Dialogues Front-End
To Telecom Gold."

Peter A. Totterdell, Paul Cooper.

Proceedings for the 1986 HCI Conference. Publishers - Cambridge Press.
1986.

[Totterdell, P.A:1986b]

"The Use of Models."

Peter A. Totterdell.

Proceedings for the 1986 Alvey Conference. July 1986.

[Totterdell, P.A:1987]

"Levels of Adaptivity in Interface Design."

P. A. Totterdell, M. A. Norman, D. P. Browne.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers -
Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Trigg, R.H:1987]

"Adaptability and Tailorability in 'NoteCards'."

Randall H. Trigg, Thomas P. Moran, Frank G. Halasz.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Tsichritzis, D.C:1988]

"Fitting Round Objects into Square Databases."

D. C. Tsichritzis, O. M. Nierstraasz.

Proceedings for the 1988 ECOOP Conference. Publishers - Springer-Verlag, New York, N.Y., U.S.A. 19.

[Tyldesley, D.A:1988]

"Employing Usability Engineering in the Development of Office Products."

D. A. Tyldesley.

The Computer Journal. 1988, Volume: 31, Number: 5.

[Vandor, S:1983]

"The Starburst User Interface: Linking Multiple Programs via Custom-Menu Software."

Steven Vandor.

Byte Magazine. December 1983.

[Verity, J.W:1987]

"The OOPS Revolution."

John W. Verity.

Datamation. May 1987.

[Vonk, R:1990]

"Prototyping - The Effective Use of CASE Technology."

Roland Vonk.

Publishers - Prentice Hall International, London, U.K. 1990.

[Waldhor, K:1987]

"Some Thesis on UNDO/REDO Commands."

Klemens Waldhor.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Warfield, R.W:1983]

"The New Interface Technology: An Introduction to Windows and Mice."

Robert W. Warfield.

Byte Magazine. December 1983.

[Warren, C:1987]

"The Role of Task Characterization in Transferring Model of Users: The Example of Engineering Design."

Clive Warren, Andy Whitefield.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Wasserman, A.I:1982]

"The Future of Programming."

Anthony I. Wasserman, Stephen Gutz.

Communications of the ACM. March 1982, Volume: 25, Number: 3.

[Wasserman, A.I:1984]

"Developing Interactive Information Systems with the User Software Engineering Methodology."

Anthony I. Wasserman.

Proceedings for the 1984 Interact Conference. Publishers - Elsevier Science Publishers B. V. (North-Holland). 1984.

[Wasserman, A.I:1989]

"An Introduction to Object Oriented Structured Design."

Anthony I. Wasserman, Peter A. Pircher, Robert J. Muller.

Internal Paper: Interactive Development Environments, Inc, San Francisco, California, USA. 1989.

[Wasserman, A.I:1990]

"The Object-Oriented Structured Design Notation for Software Design Representation."

Anthony I. Wasserman, Peter A. Pircher, Robert J. Muller.

IEEE Computer. March 1990, Volume: 23, Number: 3.

[Waterworth, J:1986]

"Interacting with Machines by Voice."

John Waterworth.

Book: The User Interface: Human Factors in Computer Based Systems.
Publishers - York University. 1986.

[Wegner, P:1989]

"Learning the Language."

Peter Wegner.

Byte Magazine. March 1989.

[Weinberg, V:1979]

"Structured Analysis."

Victor Weinberg.

Publishers - Yourdon Press, New York, N.Y., U.S.A. 1979.

[Welsh, J:1987]

"Introduction to Modula-2."

Jim Welsh, John Elder.

Publishers - Prentice Hall International, London, U.K. 1987.

[Wendel, R:1987]

"Developing Exploratory Strategies in Training: A General Approach, and Specific Example for Manual Use."

Rigas Wendel, Michael Frese.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Whitefield, A:1987]

"Models in Human Computer Interaction: A Classification with Special Reference to Their Uses in Design."

Andy Whitefield.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Whiteside, J:1987]

"The Dialect of Usability Engineering."

John Whiteside, Dennis Wixon.

Proceedings for the 1987 Interact Conference (Stuttgart). Publishers - Elsevier Science Publishers B. V. (North-Holland). September 1987.

[Wichmann, B.A:1982]

"A Comparison of Pascal and Ada."

B. A. Wichmann.

The Computer Journal. 1982, Volume: 25, Number: 2.

[Wickens, C.D:1984]

"Engineering Psychology and Human Performance."

C. D. Wickens.

Publishers - Merrill, Columbus and Ohio. 1984.

[Wiederhold, G:1986]

"View, Objects, and Database."

Gio Wiederhold.

IEEE Computer. December 1986.

[Wiest, J.D:1977]

"A Management Guide to PERT/CPM."

Jerome D. Wiest, Ferdinand K. Levy.

Publishers - Prentice Hall International, London, U.K. 1977.

[Williges, R.C:1987]

"The Use of Models in Human Computer Interface Design."

Robert C. Williges.

Ergonomics Society Lecture, Swansea, Wales. April 1987.

[Wolczko, M:1987]

"Semantics of Smalltalk 80."

Mario Wolczko.

Proceedings for the 1987 ECOOP Conference. Publishers - Springer-Verlag, New York, N.Y., U.S.A. 1987.

[Yau, S.S:1986]

"A Survey of Software Design Techniques."

Stephen S. Yau, Jeffery J.P. Tsai.

IEEE Transactions on Software Engineering. June 1986, Volume: 12, Number: 6.

[Zhao, L:1988]

"An Object Oriented Data Model for Database Modelling, Implementation and Access."

Liping Zhao, S. A. Roberts.

The Computer Journal. 1988, Volume: 31, Number: 2.

Appendix A.

Library Notebook Statements.

This appendix lists the various notebook statements collected during the library system investigation described in chapter seven. The statements are grouped and listed in subject order, and are omitted where subjects failed to complete the notebook provided.

A.1. Subject Two.

"If you are at task edit and you remain on same record, same screen format, but change to display task, any editing you have previously done on that record, e.g. changing loan categories, amending incorrect spelling, automatically reverts back to what it was before you amended it. This is OK if you remember not to do it, but it can be irritating if you forget."

"I make pointer files of records that are to be deleted. The reference numbers for these records often come to me from other people who occasionally get the reference number wrong. This can be very irritating as once in a pointer file you can only get out of it by closing that file down and starting another - no good if you want to double check a reference. It means you have to go running around looking for a vacant terminal to do a check. Also without going through a very long and complicated procedure of merging two pointer files you cannot add anything to a pointer file or remove any records from a pointer file."

"Problem with not enough terminals. As more and more of our work is based on the audit system we need more terminals. It is very difficult for some people to find a spare one to work at, thereby causing frustration and irritation. At really busy times people may come and use your terminal if you are away for half an hour or so, doing something else. This I don't mind, but you often have to wait for them to finish before you can use it."

"I find it very annoying that the system sometimes goes down, occasionally for whole or half days, and the work is waiting to be done, but you can't do it." "Another small but irritating point is the incredibly slow response time encountered occasionally. There are, on occasions, times when I've been convinced that the cursor has stuck because it has been so slow to respond."

"I have still not found a really comfortable height combination for footrest and chair. I find I get quite fidgety after a time."

"I find I have difficulty when having to look at 3 different things (e.g. the screen, a book, and a form) and take information from them (this could relate back to the above point about not sitting comfortably). I really find this a strain on my eyes more than anything else I do."

A.2. Subject Four.

"Forgetting to use q when wishing to change data sets. Logging out, and consequently having to go through full logon procedure to change data sets."

"When inputting a bibliographic reference it is easy to put the preferred form in the Name prefer box, even if it is not a name as there are no boxes differentiating other fields (apart from 'prefer'). Would be better if the field were simply 'prefer'. "

"When attempting to edit records in a pointer file, 'Help' typed at the prompt pointer file task does not tell you how to obtain the 'read from pointer file' prompt which will allow you to get into a particular pointer file. You have to remember that you must return to Task level (not pointer file task) and type md to obtain the 'read from pointer file' prompt."

"Cursor takes a long time to move when changing through a series of records to edit them."

"Very annoying when thrown back to task level with a 'MIDAS 2000' error message, and it is not exactly clear why this has happened."

"Fact that records are not in a logical order when working at a given key can be annoying."

"When attempting to edit a series of records at the same key and ask for 'move' with one of the records and then try to return to original screen of that record using =, doesn't work. Although it does work when editing records not using a key. It is necessary to change screens, or move onto next key and work backwards."

"If thrown back to task level for some reason, you need to check that what you have just edited has indeed been edited."

"Editing a record, adding a subject heading using e, Pressed return and this heading is accepted. Then changed task to d since wished to use this task for next record, pressed return before changing number in next box and the subject heading I had added was deleted."

"Changing to m in next box delete remaining numbers of reference number, but move cursor slightly to far and the format number is deleted. The message appears 'Not Found on Format File', and there is no time to correct this before you are thrown back to task level, and any editing you may have done on a record may have been ignored."

"If change record number in next box with a in task box, and a key, forget to change the task to e or d, and press return you are moved on to the next record at the key, and there is no warning of the fact that you need to change tasks to obtain the record you want."

"When searching with 'f' and change to 'a' to edit a record and then 'f' intending to move on to next record, you are thrown back to the beginning of the sequence at that key. To avoid this you need to remember to change the screen format and then change back to your original format."

"When finding a series of records the first record which appears differs on separate occasions."

A.3. Subject Five.

"Slow response times make work erratic and frustrating - can't tell whether system has registered a command and repeat it unnecessarily."

"It would be helpful if the Insert key functioned and would save time in editing."

"There seems to be an optimum concentration span - after which I cease to see errors in the text on the screen, but just continue to press keys automatically. Stuffy office seems to be a contributing factor as well."

"What do you do when the system is down, and most of your work is geared to using terminals. It can be time wasting, and unproductive."

"Do I really see an end product ? Once I've input or edited a record, it disappears into the system - doesn't seem tangible."

"Difficulty of moving between systems - Using AOLIB and then the Sirius microcomputers."

"There never seem to be enough terminals to go around."

"If I use the system to answer a readers enquiry, I feel that I am creating a mystique. They do not have access to the on-line catalogue."

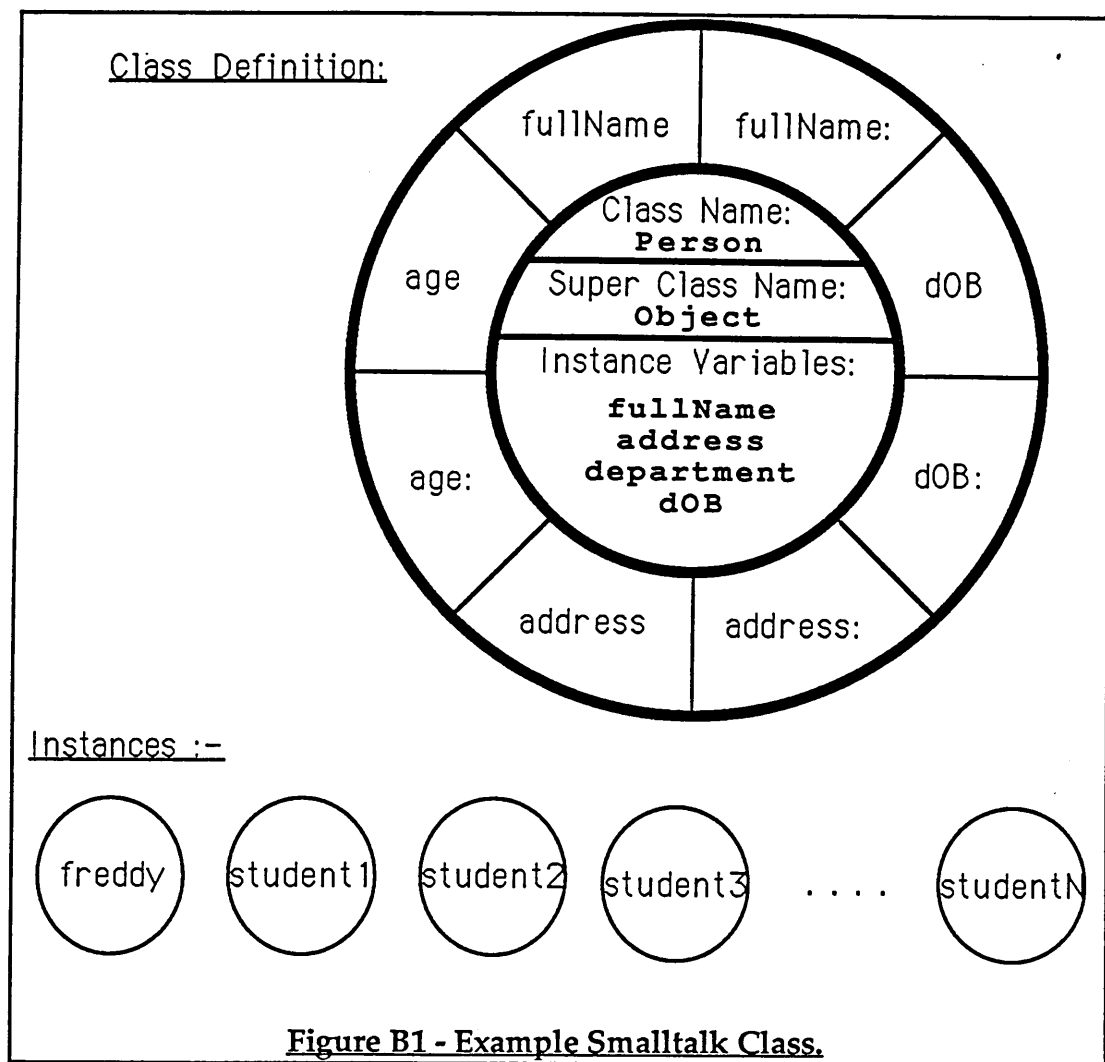
Appendix B.

An Overview of the Smalltalk 80 Programming Language and Environment.

This appendix describes the salient features of Smalltalk 80, which are relevant to the implementation work presented in chapters five, six and seven. Its content provides an overview of the applicable Smalltalk 80 concepts and terminology. For a more detailed expose of the Smalltalk 80 language and environment, the reader is directed to [Byte:1981], and books by Goldberg [Goldberg, A:1983], [Goldberg, A:1983b] and Mevel [Mevel, A:1987].

Smalltalk, object functions are known as methods. These methods have a name, and are instantiated by sending a message to an object with that name. Throughout the thesis, these messages are shown in *italics*. Internal data fields are known as instance variables (other fields known as temporary, global and Class variables also exist, but are not described here). As shown in figure B1, example instance variables include *fullName*, *address*, *department* and *dOB*; example methods include *fullName*, *fullName:*, *age*, *age:*, *dOB*, *dOB:*, *address*, and *address:*.

When a message is sent to an object, the defined method code for the message name is executed. This code may then modify internal data fields, and send messages to other objects. Upon completion, this method code also returns an object. The programmer may specify the object returned, and the default is to return the receiver object. In effect, a Smalltalk Class defines the behaviour and implementation of all the Class's instances. Each instance maintains its own unique, named instance variables, and shares the method code defined within its class with all other instances of the same Class. Whenever the method code refers to an instance variables, the instance variable belonging to the receiving instance object is inferred.



Objects are identified using a name, and messages are sent by first specifying the receiver object, followed by the message name and any necessary arguments. There are three types of messages (and therefore methods), characterised by the number of arguments required. These are Unary, Keyword, and Binary. Unary messages do not require any arguments, for example *freddy age*, *freddy address*, *freddy dOB*, and *freddy fullName* (where *freddy* is the receiver object). Keyword messages are probably the most common type, and allow arguments to be specified. The Keyword message is made up of one or more keywords preceding each argument. Each keyword terminates with a colon (':'). For example, *freddy fullName: 'Bloggs'*, and *freddy bornDay: 25 month: 4 year: 1960* (this is an example of a keyword message which allows 3 arguments, with the method name being *bornDay:month:year:*). How these arguments are interpreted by the method code depends upon the programmer's implementation. Finally, in the case of a single argument, it is possible to use a keyword made up of one or two non-alphanumeric characters, without a terminating colon. This represents

a binary message and examples include $1 + 2$ (which returns 3), and $23 > 45$ (which returns *false*).

A Smalltalk object's method code may send further messages to itself by setting the message receiver to *self*. For example, an instance method defined for the Person Class may include the expression *self age*. This effectively sends the message *age* to the receiver Person instance which is executing the defined method code.

Smalltalk implements polymorphism by allowing Sub-classes to override method code implemented in their Super-classes. For example, consider the Engine Class from appendix D, which implements an instance method called *fuelType*. In the case of an Engine instance this returns the string "Petrol". A DieselEngine Class could be implemented as a Sub-class of the Engine Class, and may also implement an instance method called *fuelType*. However, in this case the DieselEngine Class implements method code which returns the string "Diesel Fuel". Whenever the *fuelType* message is sent to a DieselEngine instance, "Diesel Fuel" is returned. If the same message is sent to an Engine instance, "Petrol" is returned instead. Smalltalk also allows this method overriding to be overridden. The overriding method code of a Sub-class may also invoke the overridden Super-class method using the message receiver *super*. This is used in an identical way to *self*. For example, the *fuelType* instance method code defined in the DieselEngine Class may include the statement *super fuelType*. This effectively invokes the method code for *fuelType* defined in the DieselEngine's Super-class. In this case it would invoke the Engine Class's *fuelType* method code and return "Petrol". If the Super-class hierarchy does not define the relevant method, an error will occur.

A Smalltalk Class defines the internal data fields and messages which are understood by all of its instances. It also defines a set of Class messages which only it (and any Sub-classes) understands. For example, the message *new*, which when sent to a Class returns an object which is a new instance of the Class. Smalltalk Classes are themselves objects (i.e. instances of another Class), and this concept makes Smalltalk a very flexible language.

Smalltalk differentiates between upper and lower case letters. This means that two methods with the same name, but different case, can exist for the same object, for example *age* and *Age*. All objects have names which act as pointers to an object, and one object may have more than one name

pointing to it. Class names are identified as starting with an uppercase letter, and there should only be one name pointer per Class. All other names should begin with a lower case letter. Only a Class object can understand a Class message (provided an appropriate method is defined), while only instances of a Class can respond to the instance messages defined by the object's Class. All instance variables must begin with a lowercase letter, and by convention, all instance methods also begin with lowercase.

Throughout the thesis unnamed instances of a Class are referred to by their Class name. For example, 'A Person understands the *age:* message' is to be interpreted as all instances of the Person Class understand the *age:* message. When reference is made to the Person Class definition itself, the term 'Person Class' is used.

Smalltalk provides the facility to define dependencies between objects in such a way that, when an object is dependent upon a second one, any alteration of the second is signalled to the first, this is known as Object Dependency. Messages are provided which allow object dependency to be created and broken. An object's existing dependencies are automatically passed on to any new objects which are made dependent upon it. Dependent objects can communicate indirectly using specific messages. These include *changed* and *changed:* which inform all dependents that an object has changed, the latter allowing arguments to be included and passed onto the dependents. Dependent objects must implement specific methods to respond to changes within their dependents. Two methods, *update* and *update:* are of particular interest. The former represents the message sent to an object when one of its dependents has used the *change* message. The latter is sent when a dependent uses the *changed:* message, and the argument is set to the argument included in the original *changed:* message.

Two important concepts arise. Firstly, an object doesn't need to know how many, or what type of objects are dependent upon it (although messages are provided to accomplish this). An object sends only one message informing of a change, or asking permission to change. Smalltalk automatically informs relevant dependents and returns the results of any inquiries. Secondly, the dependency between object A and another object B, can be made or broken by either A or B. Effectively, an interface can be made dependent upon an application without the application being affected.

The *change:* message is more specific than *change*, and it requires a single argument which is usually a method name, or identifier. It is used to notify any dependents that a specific change has occurred and the interpretation of the argument by the dependents identifies the specificity. As a consequence, any dependents are notified of the change with the message *update:* being sent to them. This message again requires a single argument, and this is automatically set to the same argument as used in the initial *changed:* message. What the dependents do when they receive this message depends upon their method code.

A Smalltalk application is made up of a set of communicating objects, each defined or represented by a set of corresponding Class descriptions. Only one message can be sent to an object at a time. The receiving object may then repeat the process, forwarding other messages to other objects within the application. At the lowest level, interaction with an application take the form of message passing with appropriate objects. This work is concerned with building user interfaces upon this level, allowing separate reconfigurable interface to be implemented for individual objects. The interface description for these objects are then attached to the appropriate Class descriptions.

Appendix C.

Object Oriented Quantitative User Model Source Code.

This appendix contains the actual Smalltalk 80 source code for the Quantitative User Model implementation discussed in chapter five.

Object subclass: #UserModel

instanceVariableNames: 'usageDict learnFormula learnFormulaString defaultApplicationIncrease defaultClassIncrease title defaultLevel2Trigger defaultLevel3Trigger usageCount errorCount errorCount2 lastUsed useSinceLastError level3Trigger level2Trigger userOverRide expertiseLevel '
classVariableNames: ''
poolDictionaries: ''
category: 'User Model'

This class represents a Quantitative User Model, which enables use of an object oriented application to be monitored for the purpose of predicting user expertise with different components of the application.

<usageDict> A dictionary containing usage information for individual classes and methods. The keys represent either class names, or combined class/method names. The value associated with each key is also a dictionary which contains the relevant information.
<learnFormulaString> A mathematical formula in the form of a string containing a block (e.g. '{x|x"x}'), which describes a users ability to learn.
<learnFormulaString> The compiled Block from <learnFormulaString>.
<defaultApplicationIncrease>, <defaultClassIncrease>, <defaultLevel2Trigger>, and <defaultLevel3Trigger> Numbers representing default values when new Classes and methods are added.
<title> String representing User Model title.
<usageCount>, <errorCount>, <errorCount2>, <lastUsed>, <useSinceLastError>, <level3Trigger>, <level2Trigger>, <userOverRide>, and <expertiseLevel> Information describing overrall Application usage. Note that this same information is stored for every class and method in the <usageDict>.

UserModel methodsFor: Addition/Deletion

addClass: aClassName

"Add new Class information"

```
| newClass |
newClass ← Dictionary new.
newClass at: #errorCount put: 0.
newClass at: #errorCount2 put: 0.
newClass at: #usageCount put: 0.
newClass at: #lastUsed put: Date today.
newClass at: #useSinceLastError put: 0.
newClass at: #applicationIncrease put: defaultApplicationIncrease.
newClass at: #level3Trigger put: defaultLevel3Trigger.
newClass at: #level2Trigger put: defaultLevel2Trigger.
newClass at: #userOverRide put: 0.
newClass at: #expertiseLevel put: nil.
usageDict at: aClassName put: newClass.
self changed: #availableClasses.
↑newClass
```

addMethod: aMethodName forClass: aClassName

"Add information for new Class method"

```
| newMethod |
usageDict at: aClassName ifAbsent: [self addClass: aClassName].
newMethod ← Dictionary new.
newMethod at: #errorCount put: 0.
newMethod at: #errorCount2 put: 0.
newMethod at: #usageCount put: 0.
newMethod at: #lastUsed put: Date today.
newMethod at: #classIncrease put: defaultClassIncrease.
newMethod at: #useSinceLastError put: 0.
newMethod at: #level3Trigger put: defaultLevel3Trigger.
newMethod at: #level2Trigger put: defaultLevel2Trigger.
newMethod at: #userOverRide put: 0.
newMethod at: #expertiseLevel put: nil.
usageDict at: (aClassName , '.' , aMethodName) asSymbol put: newMethod.
self changed: #methodsForClass:.
↑newMethod
```

removeClass: aClassName

"Remove existing Class information"

```
usageDict removeKey: aClassName ifAbsent: [].
self changed
```

removeMethod: aMethodName forClass: aClassName

"Remove existing Class method information"

```
usageDict at: aClassName ifAbsent: [self addClass: aClassName].
usageDict removeKey: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [].
self changed
```

UserModel methodsFor: Enquiry - Application

applicationErrorCount

↑errorCount

applicationErrorCount2

↑errorCount2

applicationExpertiseLevel

"Calculate Application Expertise Level"

| adjustUsage |

↑self applicationOverRidden

ifTrue: [expertiseLevel]

ifFalse:

[adjustUsage ← learnFormula value: usageCount.

adjustUsage > level3Trigger

ifTrue: [3]

ifFalse: [adjustUsage > level2Trigger

ifTrue: [2]

ifFalse: [1]]]

applicationLastUsed

↑lastUsed

applicationLevel2Trigger

↑level2Trigger

applicationLevel3Trigger

↑level3Trigger

applicationOverRidden

↑expertiseLevel notNil

applicationUsage

↑usageCount

applicationUseSinceLastError

↑useSinceLastError

UserModel methodsFor: Enquiry - Class

classApplicationIncrease: aClassName

↑(usageDict at: aClassName ifAbsent: [self addClass: aClassName])

at: #applicationIncrease

classErrorCount2: aClassName

↑(usageDict at: aClassName ifAbsent: [self addClass: aClassName])

at: #errorCount2

classErrorCount: aClassName

↑(usageDict at: aClassName ifAbsent: [self addClass: aClassName])

at: #errorCount

classExpertiseLevel: aClassName

"Calculate Class Expertise Level"

| adjustUsage class |

↑self applicationOverRidden

ifTrue: [expertiseLevel]

ifFalse: [(self classOverRidden: aClassName)

ifTrue: [(usageDict at: aClassName)

at: #expertiseLevel]

ifFalse:

[adjustUsage ← learnFormula value: ((class ← usageDict at: aClassName ifAbsent: [self addClass: aClassName]) at:

#usageCount).

adjustUsage > (class at: #level3Trigger)

ifTrue: [3]

ifFalse: [adjustUsage > (class at: #level2Trigger)

ifTrue: [2]

ifFalse: [1]]]]

classLastUsed: aClassName

↑(usageDict at: aClassName ifAbsent: [self addClass: aClassName])

at: #lastUsed

classLevel2Trigger: aClassName

↑(usageDict at: aClassName ifAbsent: [self addClass: aClassName])

at: #level2Trigger

classLevel3Trigger: aClassName

↑(usageDict at: aClassName ifAbsent: [self addClass: aClassName])

```

    at: #level3Trigger

classOverRidden: aClassName
    ↑(usageDict at: aClassName ifAbsent: [self addClass: aClassName])
    at: #expertiseLevel) notNil

classUsage: aClassName
    ↑(usageDict at: aClassName ifAbsent: [self addClass: aClassName])
    at: #usageCount

classUseSinceLastError: aClassName
    ↑(usageDict at: aClassName ifAbsent: [self addClass: aClassName])
    at: #useSinceLastError

UserModel methodsFor: Enquiry - Method

methodClassIncrease: aMethodName forClass: aClassName
    ↑(usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass: aClassName])
    at: #classIncrease

methodErrorCount2: aMethodName forClass: aClassName
    ↑(usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass: aClassName])
    at: #errorCount2

methodErrorCount: aMethodName forClass: aClassName
    ↑(usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass: aClassName])
    at: #errorCount

methodExpertiseLevel: aMethodName forClass: aClassName
    "Calculate Class Method Expertise Level"

    | adjustUsage methodDict |
    ↑self applicationOverRidden
    ifTrue: [expertiseLevel]
    ifFalse: [(self classOverRidden: aClassName)
        ifTrue: [(usageDict at: aClassName)
            at: #expertiseLevel]
        ifFalse: [(self methodOverRidden: aMethodName forClass: aClassName)
            ifTrue: [(usageDict at: (aClassName , '.' , aMethodName) asSymbol)
                at: #expertiseLevel]
            ifFalse:
                [adjustUsage ← learnFormula value: ((methodDict ← usageDict at: (aClassName , '.' , aMethodName)
asSymbol ifAbsent: [self addMethod: aMethodName forClass: aClassName]) at: #usageCount).
                adjustUsage > (methodDict at: #level3Trigger)
                ifTrue: [3]
                ifFalse: [adjustUsage > (methodDict at: #level2Trigger)
                    ifTrue: [2]
                    ifFalse: [1]]]]]]

methodLastUsed: aMethodName forClass: aClassName
    ↑(usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass: aClassName])
    at: #lastUsed

methodLevel2Trigger: aMethodName forClass: aClassName
    ↑(usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass: aClassName])
    at: #level2Trigger

methodLevel3Trigger: aMethodName forClass: aClassName
    ↑(usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass: aClassName])
    at: #level3Trigger

methodOverRidden: aMethodName forClass: aClassName
    ↑((usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass: aClassName])
    at: #expertiseLevel) notNil

methodUsage: aMethodName forClass: aClassName
    ↑(usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass: aClassName])
    at: #usageCount

methodUseSinceLastError: aMethodName forClass: aClassName
    ↑(usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass: aClassName])
    at: #useSinceLastError

UserModel methodsFor: Enquiry - General

availableClasses
    | classes |
    classes ← OrderedCollection new.
    usageDict keys do: [:aClassName | aClassName isCompound ifFalse: [classes add: aClassName]].
    ↑classes

```

```

defaultApplicationIncrease
    ↑defaultApplicationIncrease

defaultClassIncrease
    ↑defaultClassIncrease

defaultLevel2Trigger
    ↑defaultLevel2Trigger

defaultLevel3Trigger
    ↑defaultLevel3Trigger

learnFormula
    ↑learnFormulaString

methodsForClass: aClassName
    | methods |
    methods ← OrderedCollection new.
    usageDict at: aClassName ifAbsent: [↑methods].
    usageDict keys do: [:aDesc | aDesc isCompound ifTrue: [aDesc classPart = aClassName ifTrue: [methods add: aDesc selectorPart]]].
    ↑methods

title
    ↑title

UserModel methodsFor: Initialize

initialize: aName
    self initializeDefaults.
    usageDict ← Dictionary new.
    title ← aName.
    self learnFormula: '[:jjj]'.
    usageCount ← 0.
    errorCount ← 0.
    errorCount2 ← 0.
    useSinceLastError ← 0.
    level2Trigger ← defaultLevel2Trigger.
    level3Trigger ← defaultLevel3Trigger.
    userOverride ← 0.
    expertiseLevel ← nil

initializeDefaults
    self defaultClassIncrease: 0.1.
    self defaultApplicationIncrease: 0.1.
    self defaultLevel3Trigger: 50.
    self defaultLevel2Trigger: 35

resetCounts
    usageDict
        associationsDo:
            [:assoc |
                assoc value at: #usageCount put: 0.
                assoc value at: #useSinceLastError put: 0.
                assoc value at: #errorCount put: 0.
                assoc value at: #errorCount2 put: 0].
    self changed

resetErrorCounts
    usageDict
        associationsDo:
            [:assoc |
                assoc value at: #errorCount put: 0.
                assoc value at: #errorCount2 put: 0].
    self changed

resetUsage
    usageDict
        associationsDo:
            [:assoc |
                assoc value at: #usageCount put: 0.
                assoc value at: #useSinceLastError put: 0].
    self changed

UserModel methodsFor: Modification - Application

applicationErrorCount2: aNum
    errorCount2 ← aNum.
    self changed: #applicationErrorCount2

```

applicationErrorCount: aNum
errorCount ← aNum.
self changed: #applicationErrorCount

applicationExpertiseLevel: aNum
userOverRide ← 5.
expertiseLevel ← aNum.
self changed: #applicationExpertiseLevel.
self changed: #applicationOverRidden.
self changed: #classExpertiseLevel:..
self changed: #methodExpertiseLevel:forClass:

applicationLastUsed: aDate
lastUsed ← aDate.
self changed: #applicationLastUsed

applicationLevel2Trigger: aNum
level2Trigger ← aNum.
self changed: #applicationLevel2Trigger.
self changed: #applicationExpertiseLevel

applicationLevel3Trigger: aNum
level3Trigger ← aNum.
self changed: #applicationLevel3Trigger.
self changed: #applicationExpertiseLevel

applicationOverRideOff
userOverRide ← 0.
expertiseLevel ← nil.
self changed: #applicationOverRidden.
self changed: #applicationExpertiseLevel.
self changed: #classExpertiseLevel:..
self changed: #methodExpertiseLevel:forClass:

applicationUsage: aNum
usageCount ← aNum.
self changed: #applicationExpertiseLevel.
self changed: #applicationUsage

applicationUseSinceLastError: aNum
useSinceLastError ← aNum.
self changed: #applicationUseSinceLastError

UserModel methodsFor: Modification - Class

changeClass: aClassName applicationIncrease: aNum
| classDict |
classDict ← usageDict at: aClassName ifAbsent: [self addClass: aClassName].
self applicationUsage: usageCount + ((classDict at: #usageCount)
* (aNum - (classDict at: #applicationIncrease))).
classDict at: #applicationIncrease put: aNum.
self changed: #classApplicationIncrease:

changeClass: aClassName errorCount2: aNum
| classDict |
classDict ← usageDict at: aClassName ifAbsent: [self addClass: aClassName].
self applicationErrorCount2: errorCount2 - (classDict at: #errorCount2) + aNum.
classDict at: #errorCount2 put: aNum.
self changed: #classErrorCount2:

changeClass: aClassName errorCount: aNum
| classDict |
classDict ← usageDict at: aClassName ifAbsent: [self addClass: aClassName].
self applicationErrorCount: errorCount - (classDict at: #errorCount) + aNum.
classDict at: #errorCount put: aNum.
self changed: #classErrorCount:

changeClass: aClassName expertiseLevel: aNum
(usageDict at: aClassName ifAbsent: [self addClass: aClassName])
at: #userOverRide put: 5.
(usageDict at: aClassName)
at: #expertiseLevel put: aNum.
self changed: #classExpertiseLevel:..
self changed: #classOverRidden:..
self changed: #methodExpertiseLevel:forClass:

changeClass: aClassName lastUsed: aDate
(usageDict at: aClassName ifAbsent: [self addClass: aClassName])
at: #lastUsed put: aDate.
self changed: #classLastUsed:

```

changeClass: aClassName level2Trigger: aNum
  (usageDict at: aClassName ifAbsent: [self addClass: aClassName])
  at: #level2Trigger put: aNum.
  self changed: #classLevel2Trigger:.
  self changed: #classExpertiseLevel:

changeClass: aClassName level3Trigger: aNum
  (usageDict at: aClassName ifAbsent: [self addClass: aClassName])
  at: #level3Trigger put: aNum.
  self changed: #classLevel3Trigger:.
  self changed: #classExpertiseLevel:

changeClass: aClassName usage: aNum
  | classDict |
  classDict ← usageDict at: aClassName ifAbsent: [self addClass: aClassName].
  self applicationUsage: usageCount + (aNum - (classDict at: #usageCount) * (classDict at: #applicationIncrease)).
  classDict at: #usageCount put: aNum.
  self changed: #classUsage:.
  self changed: #classExpertiseLevel:

changeClass: aClassName useSinceLastError: aNum
  (usageDict at: aClassName ifAbsent: [self addClass: aClassName])
  at: #useSinceLastError put: aNum.
  self changed: #classUseSinceLastError:

changeClassOverRideOff: aClassName
  (usageDict at: aClassName ifAbsent: [self addClass: aClassName])
  at: #userOverRide put: 0.
  (usageDict at: aClassName)
  at: #expertiseLevel put: nil.
  self changed: #classOverRidden:.
  self changed: #classExpertiseLevel:.
  self changed: #methodExpertiseLevel:forClass:

setAllClassesAndMethodsExpertiseLevel: aNum
  usageDict
  associationsDo:
    [:entry |
      entry at: #userOverRide put: 5.
      entry at: #expertiseLevel put: aNum].
  self changed: #classExpertiseLevel:.
  self changed: #classOverRidden:.
  self changed: #methodExpertiseLevel:forClass:.
  self changed: #methodOverRidden:forClass:

setAllClassesAndMethodsOverRideOff
  usageDict
  associationsDo:
    [:entry |
      entry value at: #userOverRide put: 0.
      entry value at: #expertiseLevel put: nil].
  self changed: #classExpertiseLevel:.
  self changed: #classOverRidden:.
  self changed: #methodExpertiseLevel:forClass:.
  self changed: #methodOverRidden:forClass:

setAllClassesExpertiseLevel: aNum
  self availableClasses do:
    [:aClassName |
      (usageDict at: aClassName)
      at: #userOverRide put: 5.
      (usageDict at: aClassName)
      at: #expertiseLevel put: aNum].
  self changed: #classExpertiseLevel:.
  self changed: #classOverRidden:

setAllClassesOverRideOff
  self availableClasses do:
    [:aClassName |
      (usageDict at: aClassName)
      at: #userOverRide put: 0.
      (usageDict at: aClassName)
      at: #expertiseLevel put: nil].
  self changed: #classExpertiseLevel:.
  self changed: #classOverRidden:

```

UserModel methodsFor: Modification - Method

```
changeMethod: aMethodName forClass: aClassName classIncrease: aNum
```

```

| classDict methodDict |
classDict ← usageDict at: aClassName ifAbsent: [self addClass: aClassName].
methodDict ← usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass:
aClassName].
self changeClass: aClassName usage: (classDict at: #usageCount)
+ ((methodDict at: #usageCount)
* (aNum - (methodDict at: #classIncrease))).
methodDict at: #classIncrease put: aNum.
self changed: #methodClassIncrease:forClass:

```

```

changeMethod: aMethodName forClass: aClassName errorCount2: aNum
| classDict methodDict |
classDict ← usageDict at: aClassName ifAbsent: [self addClass: aClassName].
methodDict ← usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass:
aClassName].
self changeClass: aClassName errorCount2: (classDict at: #errorCount2)
- (methodDict at: #errorCount2) + aNum.
methodDict at: #errorCount2 put: aNum.
self changed: #methodErrorCount2:forClass:

```

```

changeMethod: aMethodName forClass: aClassName errorCount: aNum
| classDict methodDict |
classDict ← usageDict at: aClassName ifAbsent: [self addClass: aClassName].
methodDict ← usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass:
aClassName].
self changeClass: aClassName errorCount: (classDict at: #errorCount)
- (methodDict at: #errorCount) + aNum.
methodDict at: #errorCount put: aNum.
self changed: #methodErrorCount:forClass:

```

```

changeMethod: aMethodName forClass: aClassName expertiseLevel: aNum
(usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass: aClassName])
at: #userOverride put: 5.
(usageDict at: (aClassName , '.' , aMethodName) asSymbol)
at: #expertiseLevel put: aNum.
self changed: #methodExpertiseLevel:forClass:.
self changed: #methodOverRidden:forClass:

```

```

changeMethod: aMethodName forClass: aClassName lastUsed: aDate
(usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass: aClassName])
at: #lastUsed put: aDate.
self changed: #methodLastUsed:forClass:

```

```

changeMethod: aMethodName forClass: aClassName level2Trigger: aNum
(usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass: aClassName])
at: #level2Trigger put: aNum.
self changed: #methodLevel2Trigger:forClass:.
self changed: #methodExpertiseLevel:forClass:

```

```

changeMethod: aMethodName forClass: aClassName level3Trigger: aNum
(usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass: aClassName])
at: #level3Trigger put: aNum.
self changed: #methodLevel3Trigger:forClass:.
self changed: #methodExpertiseLevel:forClass:

```

```

changeMethod: aMethodName forClass: aClassName usage: aNum
| classDict methodDict |
classDict ← usageDict at: aClassName ifAbsent: [self addClass: aClassName].
methodDict ← usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass:
aClassName].
self changeClass: aClassName usage: (classDict at: #usageCount)
+ (aNum - (methodDict at: #usageCount) * (methodDict at: #classIncrease)).
methodDict at: #usageCount put: aNum.
self changed: #methodUsage:forClass:.
self changed: #methodExpertiseLevel:forClass:

```

```

changeMethod: aMethodName forClass: aClassName useSinceLastError: aNum
(usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass: aClassName])
at: #useSinceLastError put: aNum.
self changed: #methodUseSinceLastError:forClass:

```

```

changeMethodOverRideOff: aMethodName forClass: aClassName
(usageDict at: (aClassName , '.' , aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass: aClassName])
at: #userOverride put: 0.
(usageDict at: (aClassName , '.' , aMethodName) asSymbol)
at: #expertiseLevel put: nil.
self changed: #methodExpertiseLevel:forClass:.
self changed: #methodOverRidden:forClass:

```

```

setAllMethodsForClass: aClassName expertiseLevel: aNum

```

```

(self methodsForClass: aClassName)
do:
    [:aMethodName |
        (usageDict at: (aClassName , ':' , aMethodName) asSymbol)
            at: #userOverRide put: 5.
        • (usageDict at: (aClassName , ':' , aMethodName) asSymbol)
            at: #expertiseLevel put: aNum].
self changed: #methodExpertiseLevel:forClass:.
self changed: #methodOverRidden:forClass:

```

setAllMethodsOverRideOffForClass: aClassName

```

(self methodsForClass: aClassName)
do:
    [:aMethodName |
        (usageDict at: (aClassName , ':' , aMethodName) asSymbol)
            at: #userOverRide put: 0.
        (usageDict at: (aClassName , ':' , aMethodName) asSymbol)
            at: #expertiseLevel put: nil].
self changed: #methodExpertiseLevel:forClass:.
self changed: #methodOverRidden:forClass:

```

UserModel methodsFor: Modification - General

```

defaultApplicationIncrease: newVal
    defaultApplicationIncrease ← newVal.
    self changed: #defaultApplicationIncrease

```

```

defaultClassIncrease: newVal
    defaultClassIncrease ← newVal.
    self changed: #defaultClassIncrease

```

```

defaultLevel2Trigger: newVal
    defaultLevel2Trigger ← newVal..
    self changed: #defaultLevel2Trigger

```

```

defaultLevel3Trigger: newVal
    defaultLevel3Trigger ← newVal..
    self changed: #defaultLevel3Trigger

```

```

learnFormula: aString
    learnFormulaString ← aString.
    learnFormula ← Compiler new
        evaluate: aString
        in: nil
        to: nil
        notifying: nil
        ifFail: [].
    self changed: #learnFormula.
    self changed: #applicationExpertiseLevel.
    self changed: #classExpertiseLevel:.
    self changed: #methodExpertiseLevel:forClass:

```

```

title: aString
    title ← aString.
    self changed: #title

```

UserModel methodsFor: Usage

```

errorWithApplication
    errorCount ← errorCount + 1.
    useSinceLastError ← 0.
    userOverRide ← userOverRide = 0
        ifTrue:
            [expertiseLevel ← nil.
             0]
        ifFalse: [userOverRide - 1].
    self changed: #applicationErrorCount.
    self changed: #applicationUseSinceLastError.
    self changed: #applicationOverRidden.
    self changed: #applicationExpertiseLevel

```

```

errorWithClass: aClassName
    | classDict newTotal |
    self errorWithApplication.
    classDict ← usageDict at: aClassName ifAbsent: [classDict ← self addClass: aClassName].
    classDict at: #errorCount put: (classDict at: #errorCount)
        + 1.
    classDict at: #useSinceLastError put: 0.
    classDict at: #userOverRide put: ((newTotal ← classDict at: #userOverRide) = 0
        ifTrue:

```



```

        [classDict at: #expertiseLevel put: nil.
        0]
        ifFalse: [newTotal - 1]).
    self changed: #classErrorCount:.
    self changed: #classUseSinceLastError:.
    self changed: #classOverRidden:.
    self changed: #classExpertiseLevel:

errorWithMethod: aMethodName forClass: aClassName
    | methodDict newTotal |
    self errorWithClass: aClassName.
    methodDict ← usageDict at: (aClassName , ' ', aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass:
aClassName].
    methodDict at: #errorCount put: (methodDict at: #errorCount)
        + 1.
    methodDict at: #useSinceLastError put: 0.
    methodDict at: #userOverRide put: ((newTotal ← methodDict at: #userOverRide) = 0
        ifTrue:
            [methodDict at: #expertiseLevel put: nil.
            0]
            ifFalse: [newTotal - 1]).
    self changed: #methodErrorCount:forClass:.
    self changed: #methodOverRidden:forClass:.
    self changed: #methodUseSinceLastError:forClass:.
    self changed: #methodExpertiseLevel:forClass:

useApplication
    usageCount ← usageCount + 1.
    useSinceLastError ← useSinceLastError + 1.
    lastUsed ← Date today.
    self changed: #applicationUsage.
    self changed: #applicationUseSinceLastError.
    self changed: #applicationLastUsed.
    self changed: #applicationExpertiseLevel

useApplicationBy: aNum
    usageCount ← usageCount + aNum.
    useSinceLastError ← useSinceLastError + 1.
    lastUsed ← Date today.
    self changed: #applicationUsage.
    self changed: #applicationUseSinceLastError.
    self changed: #applicationLastUsed.
    self changed: #applicationExpertiseLevel

useClass: aClassName
    | classDict |
    classDict ← usageDict at: aClassName ifAbsent: [self addClass: aClassName].
    classDict at: #lastUsed put: Date today.
    classDict at: #usageCount put: (classDict at: #usageCount)
        + 1.
    classDict at: #useSinceLastError put: (classDict at: #useSinceLastError)
        + 1.
    self useApplicationBy: (classDict at: #applicationIncrease).
    self changed: #classLastUsed:.
    self changed: #classUseSinceLastError:.
    self changed: #classUsage:.
    self changed: #classExpertiseLevel:

useClass: aClassName by: Increase
    | classDict |
    classDict ← usageDict at: aClassName ifAbsent: [self addClass: aClassName].
    classDict at: #lastUsed put: Date today.
    classDict at: #usageCount put: (classDict at: #usageCount)
        + increase.
    classDict at: #useSinceLastError put: (classDict at: #useSinceLastError)
        + increase.
    self useApplicationBy: (classDict at: #applicationIncrease)
        * increase.
    self changed: #classLastUsed:.
    self changed: #classUseSinceLastError:.
    self changed: #classUsage:.
    self changed: #classExpertiseLevel:

useMethod: aMethodName forClass: aClassName
    | methodDict |
    methodDict ← usageDict at: (aClassName , ' ', aMethodName) asSymbol ifAbsent: [self addMethod: aMethodName forClass:
aClassName].
    methodDict at: #lastUsed put: Date today.
    methodDict at: #usageCount put: (methodDict at: #usageCount)
        + 1.

```

```
methodDict at: #useSinceLastError put: (methodDict at: #useSinceLastError)
+ 1.
self useClass: aClassName by: (methodDict at: #classIncrease).
self changed: #methodLastUsed:forClass:.
self changed: #methodUsage:forClass:.
self changed: #methodUseSinceLastError:forClass:.
self changed: #methodExpertiseLevel:forClass:
```

UserModel class

InstanceVariableNames: "

UserModel class methodsFor: Instance creation

new

↑self new: 'No Title'

new: aName

↑super new initialize: aName

Appendix D.

Smalltalk 80 Example Object Code.

This appendix contains the Smalltalk 80 source code for the example objects referred to throughout the thesis.

Contents :-

- (1) Car Class,
- (2) Chassis Class,
- (3) Department Class,
- (4) DetailedPerson Class,
- (5) DieselEngine Class,
- (6) Engine Class,
- (7) Person Class,
- (8) Sparkplug Class,
- (9) Tyre Class.

Object subclass: #Car
 instanceVariableNames: 'engine wheel1 wheel2 wheel3 wheel4 weight chassis '
 classVariableNames: ''
 poolDictionaries: ''
 category: 'Thesis examples'

A Complete Car.

Uses 6 parts :-

<engine> An instance of Engine Class.

<chassis> An instance of Chassis Class.

<wheel1> ... <wheel4> Instances of Tyre Class.

<weight> A number representing weight of the car excluding the weight of the <engine> and <chassis> parts.

Car methodsFor: Enquiry

acceleration

"Returns acceleration of car which is dependant upon total car weight and engine power.

Although not scientifically correct, this shows part dependency."

↑self totalWeight * 10 / engine power

totalWeight

"Total weight of Car"

↑weight + engine weight + chassis weight

Car methodsFor: Initialize/release

initialize

engine ← Engine new.

wheel1 ← Tyre new: 30.

wheel2 ← Tyre new: 30.

wheel3 ← Tyre new: 35.

wheel4 ← Tyre new: 35.

chassis ← Chassis new.

weight ← 1000

Car methodsFor: Modifications

weight: aNum

"Modify weight of Car itself - Not Total weight (informs of changes to wheel pressures)"

weight ← aNum.

self changed: #totalWeight.

self changed: #acceleration.

self changed: #wheel1.pressure.

self changed: #wheel2.pressure.

self changed: #wheel3.pressure.

self changed: #wheel4.pressure

Car methodsFor: Part stuff

chassis

↑chassis

chassis.chassisDescription: aDescription

"Overriding Message - inform of changes to entire wheel parts"

chassis chassisDescription: aDescription.

self changed: #totalWeight.

self changed: #acceleration.

self changed: #wheel1.

self changed: #wheel2.

self changed: #wheel3.

self changed: #wheel4

chassis.chassisNumber: aNumber

"Overriding Message - inform of changes to entire wheel parts"

chassis chassisNumber: aNumber.

self changed: #totalWeight.

self changed: #acceleration.

self changed: #wheel1.

self changed: #wheel2.

self changed: #wheel3.

self changed: #wheel4

engine

↑engine

engine.engineDescription: aDescription

"Overriding Message - informs of changes to wheel pressures"

engine engineDescription: aDescription.
self changed: #totalWeight.
self changed: #acceleration.
self changed: #wheel1.pressure.
self changed: #wheel2.pressure.
self changed: #wheel3.pressure.
self changed: #wheel4.pressure

engine.engineType: aType

"Overriding Message - informs of changes to wheel pressures"

engine engineType: aType.
self changed: #totalWeight.
self changed: #acceleration.
self changed: #wheel1.pressure.
self changed: #wheel2.pressure.
self changed: #wheel3.pressure.
self changed: #wheel4.pressure

engine.sparkplug.efficiency: aNum

"Overriding Message"

engine sparkplug.efficiency: aNum.
self changed: #acceleration

parts

↑#(chassis engine wheel1 wheel2 wheel3 wheel4)

wheel1

↑wheel1

wheel1.pressure

"Overriding Message"

↑self totalWeight * self wheel1 pressure / 1000

wheel2

↑wheel2

wheel2.pressure

"Overriding Message"

↑self totalWeight * self wheel2 pressure / 1000

wheel3

↑wheel3

wheel3.pressure

"Overriding Message"

↑self totalWeight * self wheel3 pressure / 1000

wheel4

↑wheel4

wheel4.pressure

"Overriding Message"

↑self totalWeight * self wheel4 pressure / 1000

Car class

InstanceVariableNames: "

Car class methodsFor: Instance creation

new

↑super new initialize

Object subclass: #Chassis
InstanceVariableNames: 'chassisNumber '
classVariableNames: 'DescriptionDictionary WeightDictionary '
poolDictionaries: ''
category: 'Thesis examples'

A Chassis.

<chassisNumber> a Number which represents the Chassis Type.

The following Global Dictionaries use the <chassisNumber> as a key to determine chassis characteristics :-

<DescriptionDictionary> A String representing the Chassis Description.

<WeightDictionary> A Number representing the Chassis Weight.

Chassis methodsFor: Enquiry

availableDescriptions

"Allowable Chassis Descriptions"

↑DescriptionDictionary values asOrderedCollection

chassisNumber

↑chassisNumber

chassisNumberFor: aString

"Returns Chassis Number matching description aString"

DescriptionDictionary associationsDo: [:association | aString = association value ifTrue: [↑association key]].
↑nil

description

"Description for this chassis"

↑DescriptionDictionary at: chassisNumber ifAbsent: ['Not Known']

weight

"Weight for this chassis"

↑WeightDictionary at: chassisNumber ifAbsent: [0]

Chassis methodsFor: Modifications

chassisDescription: aString

"Change Chassis so that its description matches aString."

| newType |
↑(newType ← self chassisNumberFor: aString) isNil ifFalse: [self chassisNumber: newType]

chassisNumber: aNum

"Change Chassis Number"

chassisNumber ← aNum.
self changed: #description.
self changed: #weight.
self changed: #chassisNumber

Chassis class

InstanceVariableNames: ''

Chassis class methodsFor: Global dictionarys

addDescription: aNum forType: aType

"Add new Description"

DescriptionDictionary at: aType put: aNum

addWeight: aNum forType: aType

"Add new Weight"

WeightDictionary at: aType put: aNum

Chassis class methodsFor: Initialize

Initialize

"Initialize Dictionaries"

```
DescriptionDictionary ← Dictionary new.  
WeightDictionary ← Dictionary new.  
self addDescription: 'Frame A' forType: 1.  
self addDescription: 'Frame B' forType: 2.  
self addDescription: 'Frame C' forType: 3.  
self addDescription: 'Frame D' forType: 4.  
self addDescription: 'Frame E' forType: 5.  
self addDescription: 'Frame F' forType: 6.  
self addDescription: 'Default 2' forType: 7.  
self addDescription: 'Default 3' forType: 8.  
self addDescription: 'Default 4' forType: 9.  
self addWeight: 100 forType: 1.  
self addWeight: 150 forType: 2.  
self addWeight: 200 forType: 3.  
self addWeight: 120 forType: 4.  
self addWeight: 110 forType: 5.  
self addWeight: 130 forType: 6.  
self addWeight: 160 forType: 7.  
self addWeight: 160 forType: 8.  
self addWeight: 160 forType: 9.
```

Chassis class methodsFor: Instance creation

new

↑self new: 1

new: aType

↑super new chassisNumber: aType

Object subclass: #Department
instanceVariableNames: 'departmentName head location noOfStaff '
classVariableNames: ''
poolDictionaries: ''
category: 'Thesis examples'

A Department.

<departmentName> A String representing department name.
<head> An instance of Person Class, representing who the Head is.
<location> A String representing department location.
<noOfStaff> A Number representing Number of Staff.

Department methodsFor: Enquiry

departmentName
↑departmentName

head
↑head

headName
↑head isNil
ifTrue: ['No Head']
ifFalse: [head fullName]

location
↑location

noOfStaff
↑noOfStaff

Department methodsFor: Modifications

departmentName: aString
departmentName ← aString.
self changed: #departmentName

head: aPerson
head ← aPerson.
self changed: #head

location: aString
location ← aString.
self changed: #location

noOfStaff: aNum
noOfStaff ← aNum.
self changed: #noOfStaff

takeOnMember
self noOfStaff: (noOfStaff + 1)

Department class
instanceVariableNames: ''

Department class methodsFor: Instance creation

new: aString
| a |
a ← self new departmentName: aString.
a noOfStaff: 0; head: ((Person new: 'temp')
department: a).
↑a

new: aString head: aPerson
| a |
a ← self new departmentName: aString.
a noOfStaff: 0; head: aPerson.
↑a

Person subclass: #DetailedPerson

instanceVariableNames: 'workToDo switchStatus hoursSoFar salary yearlySoFar '

classVariableNames: 'SalaryIncrease '

poolDictionaries: ''

category: 'Thesis examples'

A more Detailed Person.

<workToDo> A Number for storing a quantity of work to do.

<switchStatus> A Boolean to show how a self contained switch status is used.

<hoursSoFar> Number of hours worked by a person so far this week.

<salary> A persons hourly salary rate.

<yearlySoFar> Total salary so far this year.

<SalaryIncrease> Global Salary Increase as a percentage.

DetailedPerson methodsFor: Salary

doSalIncrease

"Perform Salary Increase"

self payRise: (salary * SalaryIncrease) // 100

endOfWeek

"Perform Week End run"

self yearlySoFar: yearlySoFar + self paySoFarThisWeek.

self hoursSoFar: 0

hoursSoFar

↑hoursSoFar

hoursSoFar: aNumber

hoursSoFar ← aNumber.

self changed: #hoursSoFar.

self changed: #paySoFarThisWeek

payRise: aNum

"Do PayRise by aNum"

self salary: salary + aNum

paySoFarThisWeek

↑salary * hoursSoFar

salary

↑salary

salary: aNumber

salary ← aNumber.

self changed: #salary.

self changed: #paySoFarThisWeek

salIncrease

↑SalaryIncrease

salIncrease: aNum

SalaryIncrease ← aNum.

self changed: #salIncrease

work: aNumber

self hoursSoFar: (hoursSoFar + aNumber)

yearlySoFar

↑yearlySoFar

yearlySoFar: aNumber

yearlySoFar ← aNumber.

self changed: #yearlySoFar.

DetailedPerson methodsFor: Switches

switchOff

"Turn switch off"

switchStatus ← false.

self changed: #switchStatus

switchOn

"Turn switch on"

switchStatus ← true.
self changed: #switchStatus

switchStatus

↑switchStatus

DetailedPerson methodsFor: Work**doWork**

"Work workToDo hours"

self work: workToDo

minus1

"Alter workToDo"

self workToDo: (workToDo - 1)

minus10

"Alter workToDo"

self workToDo: (workToDo - 10)

minus5

"Alter workToDo"

self workToDo: (workToDo - 5)

plus1

"Alter workToDo"

self workToDo: (workToDo + 1)

plus10

"Alter workToDo"

self workToDo: (workToDo + 10)

plus5

"Alter workToDo"

self workToDo: (workToDo + 5)

status1

"Initial Status for workToDo being 1"

↑true

status10

"Initial Status for workToDo being 10"

↑false

status5

"Initial Status for workToDo being 5"

↑false

workToDo

"Amount of workToDo"

↑workToDo

workToDo: aNum

"Change workToDo"

workToDo ← aNum.
self changed: #workToDo

DetailedPerson class

InstanceVariableNames: "

DetailedPerson class methodsFor: Instance Creation

```

Engine subclass: #DieselEngine
  InstanceVariableNames: ''
  classVariableNames: 'DieselDescriptionDictionary DieselEfficiencyDictionary DieselPowerDictionary
DieselSizeDictionary DieselWeightDictionary '
  poolDictionaries: ''
  category: 'Thesis examples'

```

A Diesel Engine.

This Class overrides certain Engine Class methods. In particular the sparkplug part is no longer used.

The following Global Dictionaries use the <engineType> as a key to determine other engine characteristics :-

```

<DieselDescriptionDictionary> A String representing the Diesel Engine Description.
<DieselPowerDictionary> A Number representing the Diesel Engine Power.
<DieselSizeDictionary> A Number representing the Diesel Engine Size.
<DieselWeightDictionary> A Number representing the Diesel Engine Weight.
<DieselEfficiencyDictionary> A Number representing the Diesel Engine Efficiency.

```

DieselEngine methodsFor: Enquiry

availableDescriptions

"Return available engine descriptions"

↑DieselDescriptionDictionary values asOrderedCollection

description

"Return Current Engine Description"

↑DieselDescriptionDictionary at: engineType ifAbsent: ['Not in List']

efficiency

"Return Current Engine Description"

↑DieselEfficiencyDictionary at: engineType ifAbsent: ['Not in List']

fuelType

↑'Diesel'

power

"Calculate Power. Although dependant upon sparkPlug efficiency, this value is not scientifically correct"

↑(DieselPowerDictionary at: engineType ifAbsent: [0])
* self efficiency

size

"Return Engine Size."

↑DieselSizeDictionary at: engineType ifAbsent: [0]

weight

"Return Engine Weight"

↑WeightDictionary at: engineType ifAbsent: [0]

DieselEngine methodsFor: Initialize/release

Initialize

```

engineType ← #1.
sparkplug ← nil.
running ← false

```

DieselEngine methodsFor: Modifications

engineDescription: aString

"Modify Engine Description, by updating engineType"

```

| newType |
newType ← DieselDescriptionDictionary keyAtValue: aString ifAbsent: [↑nil].
self engineType: newType

```

engineType: aType

"Modify Engine Type"

```

super engineType: aType.
self changed: #efficiency

```

DieselEngine methodsFor: Part stuff

sparkplug

"Sparkplug part"

↑'Diesel Engines do not have Sparkplugs'

DieselEngine class

instanceVariableNames: ''

DieselEngine class methodsFor: Global dictionarys

addDescription: aString forType: aType

DieselDescriptionDictionary at: aType put: aString

addEfficiency: aNum forType: aType

DieselEfficiencyDictionary at: aType put: aNum

addPower: aNum forType: aType

DieselPowerDictionary at: aType put: aNum

addSize: aNum forType: aType

DieselSizeDictionary at: aType put: aNum

addType: aType

self addDescription: 'Diesel Default' forType: aType.

self addEfficiency: 0.5 forType: aType.

self addPower: 70 forType: aType.

self addSize: 1600 forType: aType.

self addWeight: 1000 forType: aType

addWeight: aNum forType: aType

DieselWeightDictionary at: aType put: aNum

DieselEngine class methodsFor: Initialize

initialize

"Initialize Global Dictionaries.

DieselEngine initialize"

DieselSizeDictionary ← Dictionary new.

DieselWeightDictionary ← Dictionary new.

DieselPowerDictionary ← Dictionary new.

DieselEfficiencyDictionary ← Dictionary new.

DieselDescriptionDictionary ← Dictionary new.

self addType: 1.

self addDescription: 'Diesel SuperCharged Version' forType: 2.

self addEfficiency: 0.8 forType: 2.

self addWeight: 1450 forType: 2.

self addSize: 2200 forType: 2.

self addPower: 130 forType: 2.

self addDescription: 'Diesel SuperCharged Light Weight Version' forType: 3.

self addEfficiency: 0.9 forType: 3.

self addWeight: 1600 forType: 3.

self addSize: 1999 forType: 3.

self addPower: 115 forType: 3.

self addDescription: 'Diesel Cheapo Boring Efficient Engine' forType: 4.

self addEfficiency: 0.7 forType: 4.

self addWeight: 1300 forType: 4.

self addSize: 1500 forType: 4.

self addPower: 58 forType: 4

Object subclass: #Engine

instanceVariableNames: 'engineType sparkplug running '

classVariableNames: 'DescriptionDictionary PowerDictionary SizeDictionary WeightDictionary '

poolDictionaries: ''

category: 'Thesis examples'

An Engine.

Uses a single part called sparkplug, which should be an instance of the Sparkplug Class.

<engineType> a Number which represents the Engine Type.

<sparkplug> Sparkplug part.

<running> A Boolean representing whether the Engine is running or not.

The following Global Dictionaries use the <engineType> as a key to determine other engine characteristics :-

<DescriptionDictionary> A String representing the Engine Description.

<PowerDictionary> A Number representing the Engine Power.

<SizeDictionary> A Number representing the Engine Size.

<WeightDictionary> A Number representing the Engine Weight.

Engine methodsFor: Engine state

engineState

"Enquire on Engine State"

↑running

ifTrue: [#running]

ifFalse: [#notRunning]

startUp

"Start Up Engine"

running ← true.

self changed: #engineState

switchOff

running ← false.

self changed: #engineState

switchOn

running ← true.

self changed: #engineState

Engine methodsFor: Enquiry

availableDescriptions

"Return available engine descriptions"

↑DescriptionDictionary values asOrderedCollection

description

"Return Current Engine Description"

↑DescriptionDictionary at: engineType ifAbsent: ['Not in List']

engineType

↑engineType

fuelType

↑'Petrol'

power

"Calculate Power. Although dependant upon sparkPlug efficiency, this value is not scientifically correct"

↑(PowerDictionary at: engineType ifAbsent: [0])

* sparkplug efficiency

size

"Return Engine Size."

↑SizeDictionary at: engineType ifAbsent: [0]

weight

"Return Engine Weight"

↑WeightDictionary at: engineType ifAbsent: [0]

Engine methodsFor: Initialize/release

initialize

```
engineType ← #1.  
sparkplug ← Sparkplug new.  
running ← false
```

Engine methodsFor: Interface

eTypeMessage

"Message for inputing new engine type"

```
↑'enter new engine type'
```

Engine methodsFor: Modifications

engineDescription: aString

"Modify Engine Description, by updating engineType"

```
| newType |  
newType ← DescriptionDictionary keyAtValue: aString ifAbsent: [↑nil].  
self engineType: newType
```

engineType: aType

"Modify Engine Type"

```
engineType ← aType.  
self changed: #engineType.  
self changed: #description.  
self changed: #power.  
self changed: #size.  
self changed: #weight
```

Engine methodsFor: Part stuff

parts

"Return list of messages which return parts"

```
↑#{sparkplug }
```

sparkplug

"Sparkplug part"

```
↑sparkplug
```

sparkplug.efficiency: aNum

"Overriding Part message"

```
sparkplug efficiency: aNum.  
self changed: #power
```

Engine class

instanceVariableNames: "

Engine class methodsFor: Global dictionarys

addDescription: aString forType: aType

DescriptionDictionary at: aType put: aString

addPower: aNum forType: aType

PowerDictionary at: aType put: aNum

addSize: aNum forType: aType

SizeDictionary at: aType put: aNum

addType: aType

```
self addDescription: 'Default' forType: aType.  
self addPower: 60 forType: aType.  
self addSize: 1300 forType: aType.  
self addWeight: 300 forType: aType
```

addWeight: aNum forType: aType

WeightDictionary at: aType put: aNum

Engine class methodsFor: Initialize

Initialize

"Initialize Global Dictionaries"

```
SizeDictionary ← Dictionary new.  
WeightDictionary ← Dictionary new.  
PowerDictionary ← Dictionary new.  
DescriptionDictionary ← Dictionary new.  
self addType: 1.  
self addDescription: 'SuperCharged Version' forType: 2.  
self addWeight: 450 forType: 2.  
self addSize: 1999 forType: 2.  
self addPower: 130 forType: 2.  
self addDescription: 'SuperCharged Light Weight Version' forType: 3.  
self addWeight: 350 forType: 3.  
self addSize: 1999 forType: 3.  
self addPower: 115 forType: 3.  
self addDescription: 'Cheapo Boring Efficient Engine' forType: 4.  
self addWeight: 300 forType: 4.  
self addSize: 1200 forType: 4.  
self addPower: 58 forType: 4
```

Engine class methodsFor: Instance creation

new

↑super new initialize

new: aType

↑self new engineType: aType

Object subclass: #Person

InstanceVariableNames: 'fullName address department dOB '

classVariableNames: ''

poolDictionaries: ''

category: 'Thesis examples'

A Simple Person.

<fullName> A String representing Full Name.

<address> A String representing the Address.

<department> An instance of the Department Class, representing what department the person belongs to.

<dOB> A Date representing the persons Date Of Birth.

Person methodsFor: Enquiry

address

↑address

age

"age in days"

↑Date today subtractDate: dOB

department

↑department

dOB

↑dOB

fullName

↑fullName

Person methodsFor: Modifications

address: aString

address ← aString.

self changed: #address

age: dayCount

dOB ← Date today subtractDays: dayCount.

self changed: #dOB.

self changed: #age

department: aDept

department ← aDept.

self changed: #department

dOB: newDate

dOB ← newDate.

self changed: #dOB.

self changed: #age

fullName: aString

fullName ← aString.

self changed: #fullName

Person methodsFor: Part stuff

parts

"Return list of messages which return parts. Although department is not a real part, this allows safe access"

↑#(department)

Person class

InstanceVariableNames: ''

Person class methodsFor: Instance creation

new

↑self new: 'No Name'

new: aName

↑super new fullName: aName; address: 'Not Known'; department: nil; dOB: (Date today)

Object subclass: #Sparkplug
InstanceVariableNames: 'name efficiency '
classVariableNames: ''
poolDictionaries: ''
category: 'Thesis examples'

A SparkPlug

<efficiency> A number representing the sparkPlug efficiency (0.0 - 1.0).
<name> A String representing name of sparkPlug.

Sparkplug methodsFor: Enquiry

efficiency
↑efficiency

name
↑name

Sparkplug methodsFor: Modifications

efficiency: aNum
efficiency ← aNum.
self changed: #efficiency

name: aString
name ← aString.
self changed: #name

Sparkplug class InstanceVariableNames: ''

Sparkplug class methodsFor: Instance creation

new
↑self new: 'Default'

new: aName
↑(super new) name: aName; efficiency: 0.85

Object subclass: #Tyre
InstanceVariableNames: 'volume '
classVariableNames: ''
poolDictionaries: ''
category: 'Thesis examples'

A Tyre

<volume> A number representing amount of air in Tyre

Tyre methodsFor: Enquiry

inflate: aNum
self volume: volume + aNum

pressure
"Calculate pressure for unloaded tyre, Not Physically correct, but shows
example of Part Message overriding"

↑volume * 10

volume
↑volume

volume: aNum
volume ← aNum.
self changed: #pressure.
self changed: #volume

Tyre class
InstanceVariableNames: ''

Tyre class methodsFor: Instance creation

new
↑self new: 500

new: aNum
↑super new volume: aNum

Appendix E.

Extended Backus Naur Formats.

This appendix contains the Extended Backus Naur Formats, for the various interface description languages described in chapter six. These Formats make use of the actual Smalltalk 80 EBNF [Goldberg, A:1983]. The Non-terminal `<SmalltalkExpression>` provides this link to the Smalltalk 80 EBNF, and represents a correct Smalltalk 80 expression.

E.1. General Syntax.

`<UpperCaseLetter>` = "A" | "B" | "C" | | "Z".
`<LowerCaseLetter>` = "a" | "b" | "c" | | "z".
`<Letter>` = `<UpperCaseLetter>` | `<LowerCaseLetter>`.
`<Space>` = " ".
`<FullStop>` = ".".
`<SemiColon>` = ";".
`<nil>` = "nil".
`<Digit>` = "0" | "1" | "2" | | "9".
`<Integer>` = `<Digit>` {`<Digit>`}.
`<Real>` = `<Integer>` `<FullStop>` `<Integer>`.
`<IntegerArray>` = "(" {`<Integer>` `<Space>`} ")".
`<SmalltalkString>` = "\"" {`<Letter>` | `<Digit>` | `<Space>`} "\"".
`<SmalltalkWord>` = `<Letter>` {`<Letter>` | `<Digit>`}.
`<SmalltalkSymbol>` = "#" `<SmalltalkWord>`.
`<SmalltalkClass>` = `<UpperCaseLetter>` {`<SmalltalkWord>`}.

`<SmalltalkExpression>` = *This represents any valid Smalltalk 80 Expression.*

E.2. External PVC Slot Description.

`<ArgumentsRequired>` = "MsgArgs" | "NoMsgArgs" | "NilMsgArgs".
`<ArgumentCount>` = `<IntegerArray>`.
`<Includer>` = "With" | "Without".
`<MessageDescription>` = `<Includer>` `<Space>` `<ArgumentCount>` `<Space>` `<ArgumentsRequired>`.
`<LiteralType>` = "Literal" `<Space>` `<SmalltalkClass>`.
`<MessageType>` = "Message" `<Space>` `<MessageDescription>`.

<AnyType> = "Any" <Space> <SmalltalkClass> <Space>
 <MessageDescription>.
 <SlotHeader> = "SlotDescription ^#".
 <SlotType> = <LiteralType> | <MessageType> | <AnyType>.
 <SlotDefault> = <SmalltalkString>.
 <SlotDirection> = "Input" | "Output" | "IO".
 <SlotTitle> = <SmalltalkString>.
 <SlotName> = <SmalltalkWord>.
 <SlotDefinition> = "(" <SlotDirection> <Space> <SlotType> ")".
 <SlotDescription> = "(" <SlotName> <Space> <SlotTitle> <Space>
 <SlotDefinition> ")" [<Space> <SlotDefault>] [<Space> "MultiSlot"].
 <SlotDescriptionGroup> = "(" <SlotDescription> {<Space>
 <SlotDescription>} ")".
 <ExternalSlotDescription> = <SlotHeader> <SlotDescriptionGroup>.

E.2. Part PVC Description.

<Links> = <IntegerArray>.
 <PPVCLinks> = "aPPVC addIPVCLinks:" <Space> <Links>.
 <PPVCName> = <SmalltalkWord>.
 <PPVCHheader> = <PPVCName> ": aPPVC".
 <PPVCDescription> = <PPVCHheader> <PPVCBody>.
 <LessThanOrEqualOne> = "1" | ("0" <FullStop> <Integer>).
 <XCoordinate> = <LessThanOrEqualOne>.
 <YCoordinate> = <LessThanOrEqualOne>.
 <PVCScreenPosition> = <XCoordinate> "@" <YCoordinate>.
 <TopLeftCornerPosition> = <PVCScreenPosition>.
 <BottomRightCornerPosition> = <PVCScreenPosition>.
 <SubPVCPosition> = "(" <TopLeftCornerPosition> <Space> "corner:"
 <Space> <BottomRightCornerPosition> ")".
 <AssignSlotArgument> =
 <AssignSlotName> = <SmalltalkSymbol>.
 <AssignSlotType> = "#Literal" | "#Message".
 <AssignSlotValue> = <SmalltalkSymbol> | <nil> | <SmalltalkExpression>.
 <AssignSlot> = "slot:" <Space> <AssignSlotName> <Space> "value:"
 <Space> <AssignSlotValue> <Space> "type:" <Space> <AssignSlotType>.
 <MultiSlotType> = "Literal" | "Message".
 <MultiSlotVal> = "add:" <Space> <AssignValue>.
 <MultiSlotValues> = <MultiSlotVal> {<SemiColon> <Space>
 <MultiSlotVal>}.

<AssignMultiSlotValues> = "((OrderedCollection new)" <Space>
 <MultiSlotValues> ")".
 <AssignMultiSlotTypes> = "#Literal" | "#(" <MultiSlotType> {<Space>
 <MultiSlotType>} ")".
 <AssignMultiSlot> = "multiSlot:" <Space> <AssignSlotName> <Space>
 "values:" <Space> <AssignMultiSlotValues> <Space> "types:" <Space>
 <AssignMultiSlotTypes>.
 <SingleSlotArg> = <SmalltalkExpression>.
 <MultiSlotArgVal> = "add:" <Space> <SingleSlotArg>.
 <MultiSlotArgValues> = <MultiSlotArgVal> {<SemiColon> <Space>
 <MultiSlotArgVal>}.
 <MultiSlotArgs> = "((OrderedCollection new)" <Space>
 <MultiSlotArgValues> ")".
 <SlotArguments> = <MultiSlotArgs> | <SingleSlotArg>.
 <AssignSlotArgument> = 'slotArgs:' <Space> <AssignSlotName> <Space>
 "value:" <Space> <SlotArguments>.
 <SlotAssignment> = <AssignSlot> | <AssignMultiSlot> |
 <AssignSlotArgument>.
 <SlotValueList> = <Space> | <SlotAssignment> {<SemiColon> <Space>
 <SlotAssignment> }.
 <PPVCInstantiationCode> = "^((" <SmalltalkClass> <Space> "new)" <Space>
 <SlotValueList> ")".
 <SubPPVCName> = "new" | "default" | <SmalltalkSymbol>.
 <SubPPVC> = "aPPVC addPPVC:" <Space> <SubPPVCName> <Space>
 "partMsg:" <Space> <SmalltalkSymbol> <Space> "variablePPVC:
 #NotYetUsed at:" <Space> <SubPVCPosition> <FullStop>.
 <SubIPVC> = "aPPVC addSubView:" <Space> <PVCInstantiationCode>
 <Space> "window: (0 @ 0 extent: 1 @ 1) viewport:" <Space>
 <SubPVCPosition> <FullStop>.
 <SubPVCCode> = {<SubPPVC> | <SubIPVC>}.
 <PPVCBody> = "aPPVC isNil ifTrue: [" <PPVCInstantiationCode> "]" ifFalse:
 [" <SubPVCCode> <Space> <PVCLinks> "]".

E.2. Extended Lean Cuisine Description.

<ELCName> = <SmalltalkWord>.
 <MenemeStatus> = "true" | "false".
 <MenemeName> = <Letter> {<Letter> | <Digit> | <Space>}.
 <OnMethod> = <SmalltalkWord>.
 <OffMethod> = <SmalltalkWord>.

<SelectMethod> = <SmalltalkWord>.
 <Terminator> = <Bistable> | <MonoStable>.
 <NonTerminator> = <RealMEMenu> | <RealMCMenu> |
 <VirtualMEMenu> | <VirtualMCMenu>.
 <Meneme> = "(" (<Terminator> | <NonTerminator>) ")".
 <Bistable> = "Bistable" <Space> <OnMethod> <Space> <OffMethod>
 <Space> <MenemeStatus> <Space> <MenemeName>.
 <Monostable> = "Monostable" <Space> <SelectMethod> <Space>
 <MenemeName>.
 <VirtualMEMenu> = "VirtualMEMenu" <Space> <MenemeStatus>
 <Space> <MenemeName> "[" {<Meneme> <Space>} "]".
 <VirtualMCMenu> = "VirtualMCMenu" <Space> <MenemeStatus>
 <Space> <MenemeName> "[" {<Meneme> <Space>} "]".
 <RealMEMenu> = "RealMEMenu" <Space> <MenemeStatus> <Space>
 <MenemeName> "[" {<Meneme> <Space>} "]".
 <RealMCMenu> = "RealMCMenu" <Space> <MenemeStatus> <Space>
 <MenemeName> "[" {<Meneme> <Space>} "]".
 <ELCDescription> = <RealMCMenu> | <RealMEMenu>.
 <ELCMethodCode> = "^ (Menu onString:" <Space> <ELCDescription> ")".
 <ELCDescriptionMethod> = <ELCName> <ELCMethodCode>.

Appendix F.

Interaction Pluggable View Controller Library.

This appendix presents a short description of each example IPVC implemented as part of this research project. For each IPVC, the relevant External Slot Description is listed, and a picture is also included to show what the IPVC looks like while it is being used.

Contents :-

- (1) Bar Chart IPVC,
- (2) Button IPVC,
- (3) Graph IPVC,
- (4) Horizontal Slider IPVC,
- (5) List IPVC
- (6) String Editor IPVC,
- (7) Switch IPVC with Self Contained Status,
- (8) Switch IPVC with Status Contained in Model,
- (9) Text Editor IPVC,
- (10) Vertical Slider IPVC,
- (11) View and Revise General Purpose IPVC,
- (12) View and Revise String IPVC,
- (13) View Only General Purpose IPVC,
- (14) View Only String IPVC,
- (15) Special PPVC PPVCUserModelView (Slot Descriptions only, as this looks identical to a normal PPVC blank window).

SlotDescription

*BARChart IPVC.

This IPVC allows numeric values to be displayed using a Bar Chart.
Two Multiple Linkage slots are used for determining bar value and bar title.
The following Linkage slots are used :-

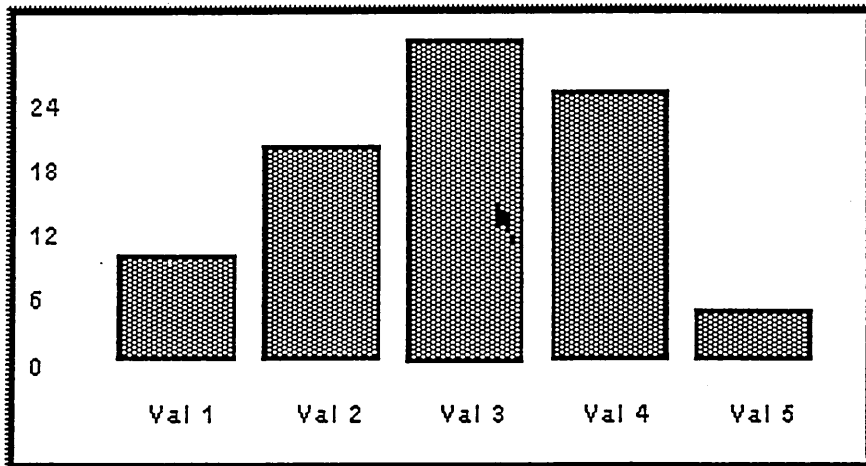
<label> Used to determine Chart title (Not actually used as no title is displayed).

<barValues> Used to determine Bar values.

<barTitles> Used to determine Bar titles.

<yBM> Used to determine Interaction menu, if any."

```
↑#( (label 'Title of Chart' (input Any String with (0 ) noMsgArgs ) ""No Label"" )  
    (barValues 'Bar Values' (input Any Number with (0 ) noMsgArgs ) '0' multiSlot )  
    (barTitles 'Bar Labels' (input Literal String ) ""No Label"" multiSlot )  
    (yBM 'yBM' (input Any Menu with (0 ) noMsgArgs ) ) )
```



IPVCButtonView class methodsFor: slot definitions

SlotDescription

"BUTTON IPVC

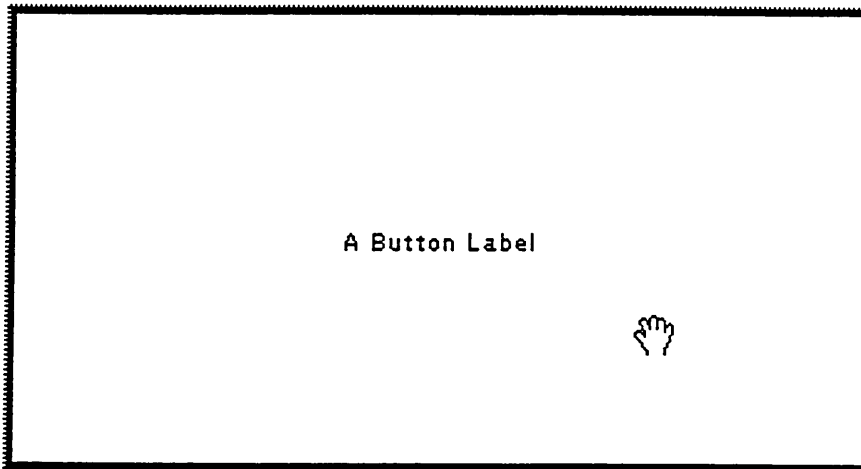
An IPVC which displays a button, and sends a specific message whenever it is pressed.

<label> Determine button label.

<switchPress> Determines what message is sent when button is pressed.

<yBM> Used to determine Interaction menu, if any."

```
↑#( (label 'Label' (Input Any String with (0 ) noMsgArgs ) "" "")  
    (switchPress 'Switch Press' (Output Message without () msgArgs ) )  
    (yBM 'YBM' (Input Any Menu with (0 ) noMsgArgs ) ) )
```



SlotDescription

*GRAPH IPVC

IPVC for displaying a formula as a graph form. The formula must be of the format '[:x|x]', and may be of Class String, or BlockContext. The X and Y axis ranges may also be set, along with the unit size.

<curveBlock> determines curve formula.

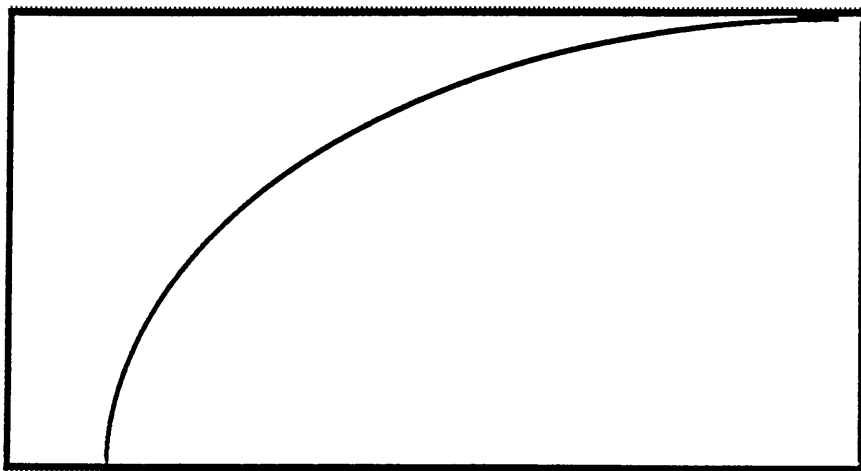
<minX>, <maxX>, <minY>, <maxY> determine axis ranges.

<step> determines unit size for X axis."

```

†#( (curveBlock 'Curve Block' (Input Any String with (0 ) noMsgArgs ) ""[:x|x]" )
    (minX 'Minimum X' (Input Any Number with (0 ) noMsgArgs ) '0')
    (maxX 'Maximum X' (Input Any Number with (0 ) noMsgArgs ) '100')
    (minY 'Minimum Y' (Input Any Number with (0 ) noMsgArgs ) '0')
    (maxY 'Maximum Y' (Input Any Number with (0 ) noMsgArgs ) '100')
    (step 'Step' (Input Any Number with (0 ) noMsgArgs ) '5')
    (yBM 'yBM' (Input Any Menu with (0 ) noMsgArgs ) ) )

```



IPVCHorizontalSliderView class methodsFor: slot definitions

SlotDescription

"HORIZONTAL SLIDER IPVC

An IPVC which displays numeric values using a graphical slider. The current value can also be modified by moving the slider.

Two types are available, namely Vertical and Horizontal slider.

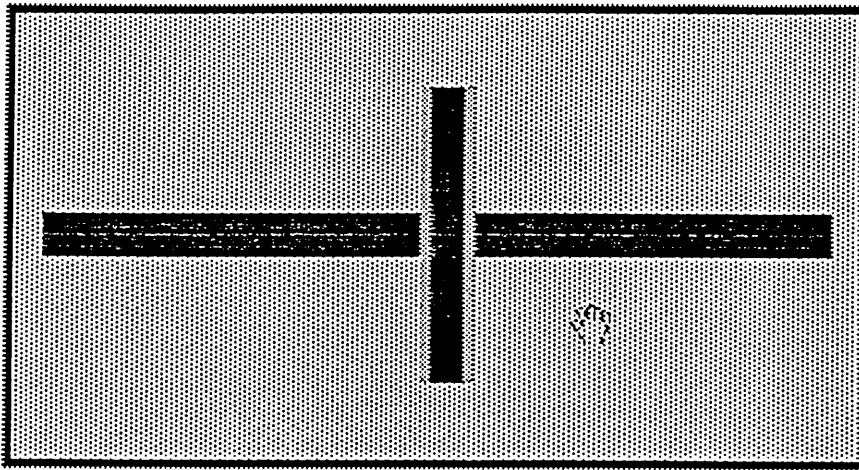
<sliderValue> Determines current value being displayed.

<sliderUpdate> A message for informing of modified value.

<sliderRange> An Interval which determines the slider range, and unit size.

<yBM> Used to determine Interaction menu, if any."

```
↑#( (sliderValue 'Slider Current Value' (Input Any Number with (0 ) noMsgArgs ) '50')
    (sliderUpdate 'Updating Model' (Output Message with (1 ) noMsgArgs ) )
    (sliderRange 'Slider Range (as Interval)' (Input Any Interval with (0 ) noMsgArgs ) '1 to: 100 by: 10' )
    (yBM 'YBM' (Input Any Menu with (0 ) noMsgArgs ) ) )
```



IPVCListView class methodsFor: slot definitions

SlotDescription

*LIST IPVCL

Used to display, and select from, a choice of options contained in an object of Class Collection. These options are displayed in a list, and the current selection is highlighted. When a new selection is made, the model is informed appropriately. Scrolling is used to move up and down the available list of options.

<currentItem> A message which returns the current selected item in the option list. <newSelection> A message which informs of a new selection.

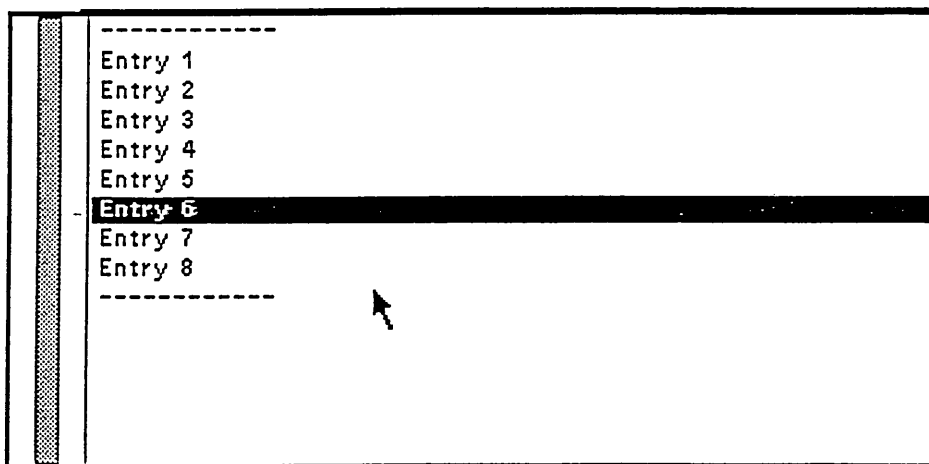
<itemList> This returns the list of available options.

<stringPrint> A boolean which determines whether the options are printed using the 'printString' method.

<oneItem> A boolean which determines whether the option list only contains oneItem.

<yBM> Used to determine Interaction menu, if any."

```
↑#( (currentItem 'Current Selected Item' (Input Message with (0 ) noMsgArgs ) )
    (newSelection 'Informing Model of new selection' (Output Message with (1 ) noMsgArgs ) )
    (itemList 'Item List' (Input Any Symbol with (0 ) noMsgArgs ) '#()')
    (stringPrint 'Print as String' (Input Literal Boolean ) )
    (oneItem 'One Item' (Input Literal Boolean ) )
    (yBM 'YBM' (Input Any Menu with (0 ) noMsgArgs ) ) )
```



SlotDescription

"STRING EDITOR IPVC

An IPVC which allows Strings to be displayed and modified. The value is displayed in an editor box, and may be modified accordingly. The new value may be accepted by choosing an option from the selection menu (Yellow Button), and the model is informed of the new value using a linkage slot. The implementor must therefore define a suitable value for the <yBM> linkage slot. Various Methods are provided for use in such a menu, and the implementor is directed to the method 'localMenuItems' implemented in the IPVCEditorIC Class.

<inputString> String value to be displayed.

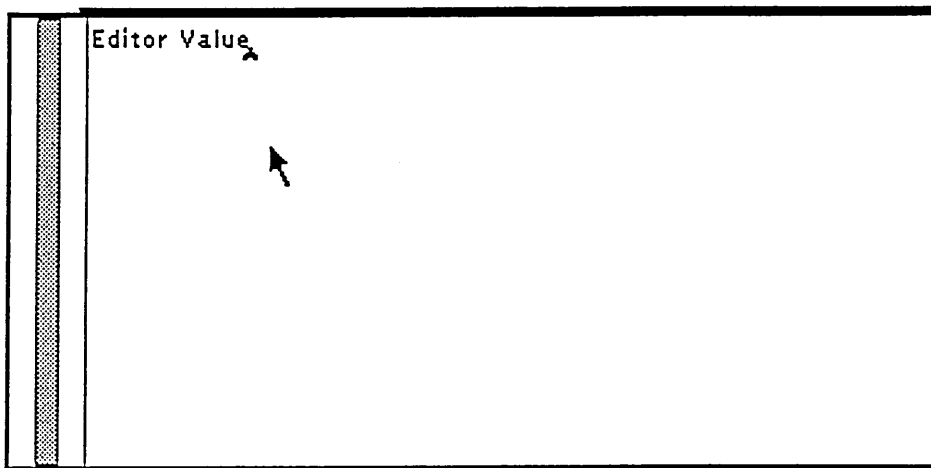
<modelUpdate> Message used to inform model of new value.

<yBM> Used to determine Interaction menu, if any."

```

+#+( (inputString 'Display Value' (Input Any String with (0 ) noMsgArgs ) ""Not Specified" )
      (modelUpdate 'Updating Model' (Output Message with (1 ) noMsgArgs ) )
      (yBM 'yBM' (Input Any Menu with (0 ) noMsgArgs ) ) )

```



SlotDescription

*SWITCH IPVC (SELF CONTAINED STATUS)

An IPVC which displays a switch with a specific title. The status of the switch is maintained within the IPVC, and its initial value is determined by a linkage slot. The switch is highlighted when on, and normal when off. The user may switch on and off by pressing the mouse button while over the switch. Depending upon whether the switch is switched on, or off, one of two message is sent to the model, as determined by linkage slots.

<label> Determine switch label.

<initialStatus> Determines Initial switch status.

<switchOff> Message sent when switching off.

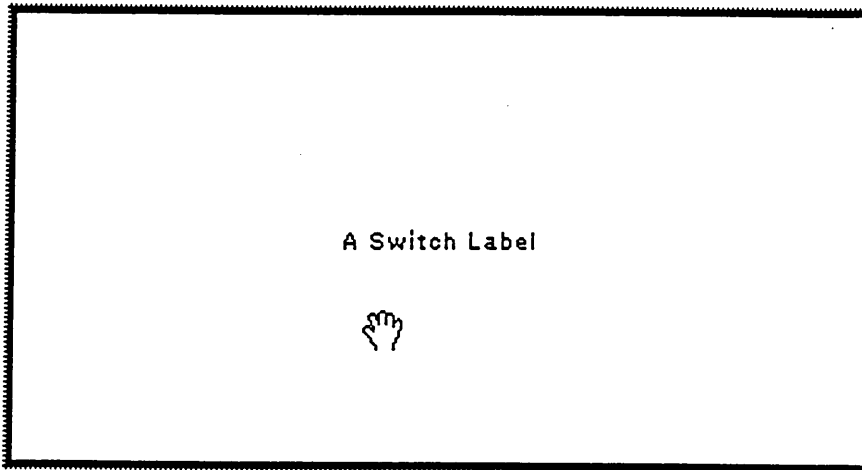
<switchOn> Message sent when switching on.

<yBM> Used to determine Interaction menu, if any."

```

+ #( (label 'Label' (Input Any String with (0 ) noMsgArgs ) "" "")
      (initialStatus 'Initial Switch State' (Input Any Boolean with (0 ) noMsgArgs ) 'true')
      (switchOff 'Switching OFF' (Output Message without () msgArgs ) )
      (switchOn 'Switching ON' (Output Message without () msgArgs ) )
      (yBM 'YBM' (Input Any Menu with (0 ) noMsgArgs ) ) )

```



SlotDescription

*SWITCH IPVC (STATUS IN MODEL)

An IPVC which displays a switch with a specific title. The status of the switch is maintained by the model, and is determined by a linkage slot. The switch is highlighted when on, and normal when off. The user may switch on and off by pressing the mouse button while over the switch. Depending upon whether the switch is switched on, or off, one of two message is sent to the model, as determined by linkage slots.

<label> Determine switch label.

<status> Determines switch status.

<switchOff> Message sent when switching off.

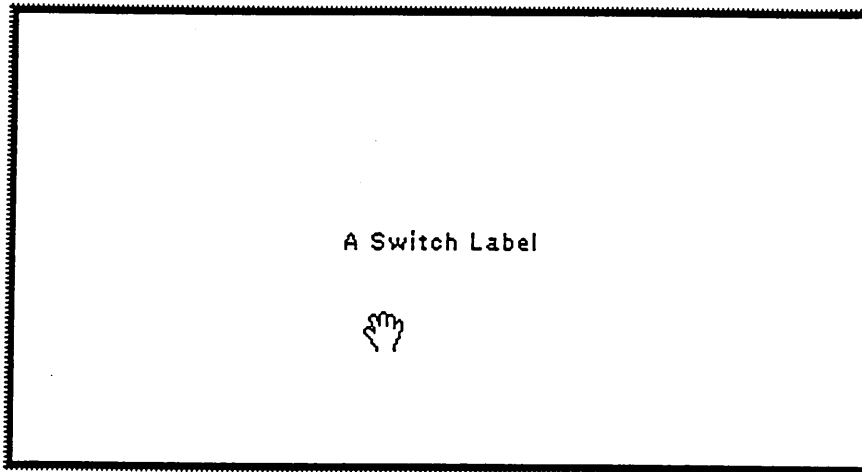
<switchOn> Message sent when switching on.

<yBM> Used to determine Interaction menu, if any."

```

*#( (label 'Label' (Input Any String with (0) noMsgArgs) "" "")
    (status 'Switch State' (Input Message Boolean with (0) noMsgArgs) )
    (switchOff 'Switching OFF' (Output Message without () msgArgs) )
    (switchOn 'Switching ON' (Output Message without () msgArgs) )
    (yBM 'YBM' (Input Any Menu with (0) noMsgArgs) ) )

```



SlotDescription

"TEXT EDITOR IPVC

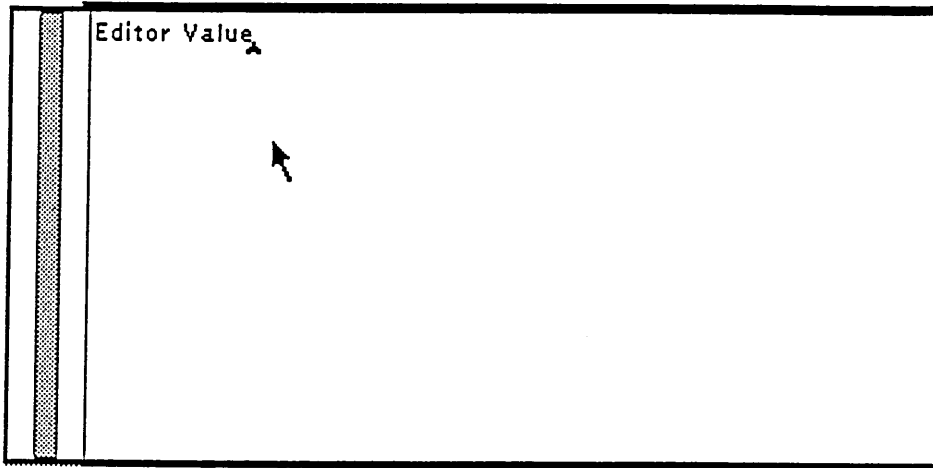
An IPVC which allows Text to be displayed and modified. The value is displayed in an editor box, and may be modified accordingly. The new value may be accepted by choosing an option from the selection menu (Yellow Button), and the model is informed of the new value using a linkage slot. The implementor must therefore define a suitable value for the <yBM> linkage slot. Various Methods are provided for use in such a menu, and the implementor is directed to the method 'localMenuItems' implemented in the IPVCTextViewIC Class.

<inputText> String value to be displayed.

<modelUpdate> Message used to inform model of new value.

<yBM> Used to determine Interaction menu, if any."

```
↑#( (inputText 'Display Value' (Input Any Text with (0 ) noMsgArgs ) 'Text new' )
    (modelUpdate 'Updating Model' (Output Message with (1 ) noMsgArgs ) )
    (yBM 'yBM' (Input Any Menu with (0 ) noMsgArgs ) ) )
```



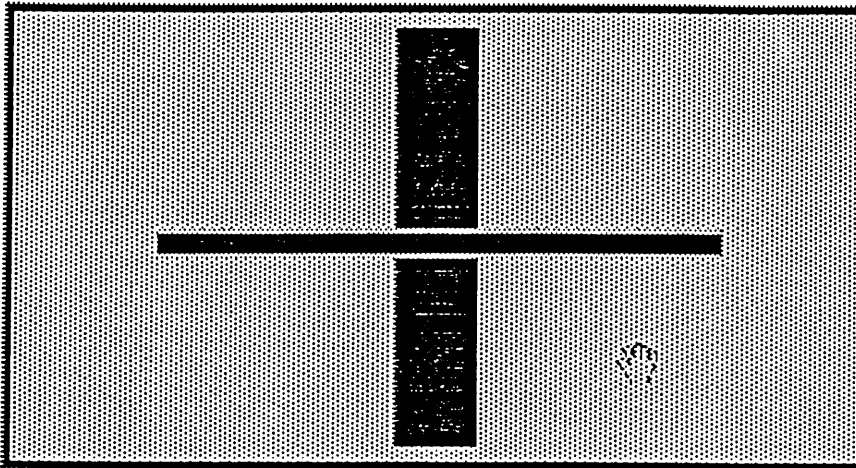
SlotDescription

"VERTICAL SLIDER IPVC

An IPVC which displays numeric values using a graphical slider. The current value can also be modified by moving the slider.
Two types are available, namely Vertical and Horizontal slider.

<sliderValue> Determines current value being displayed.
<sliderUpdate> A message for informing of modified value.
<sliderRange> An Interval which determines the slider range, and unit size.
<yBM> Used to determine Interaction menu, if any."

↑#((sliderValue 'Slider Current Value' (Input Any Number with (0) noMsgArgs) '50')
(sliderUpdate 'Updating Model' (Output Message with (1) noMsgArgs))
(sliderRange 'Slider Range (as Interval)' (Input Any Interval with (0) noMsgArgs) '1 to: 100 by: 10')
(yBM 'YBM' (Input Any Menu with (0) noMsgArgs)))



SlotDescription

"GENERAL PURPOSE VIEW AND REVISE IPVC

A general purpose IPVC which allows any type of value to be displayed and modified. The value displayed is determined using the printString method in conjunction with a linkage slot. The value is displayed in a box, and may be prefixed with a specific message. The value may be modified by pressing the mouse button while over the box. A prompt is then given to enter the new value, and the model is informed of the new value using another linkage slot.

<label> Prefix displayed in front of value.

<displayValue> Value to be displayed.

<modelUpdate> Message used to inform model of new value.

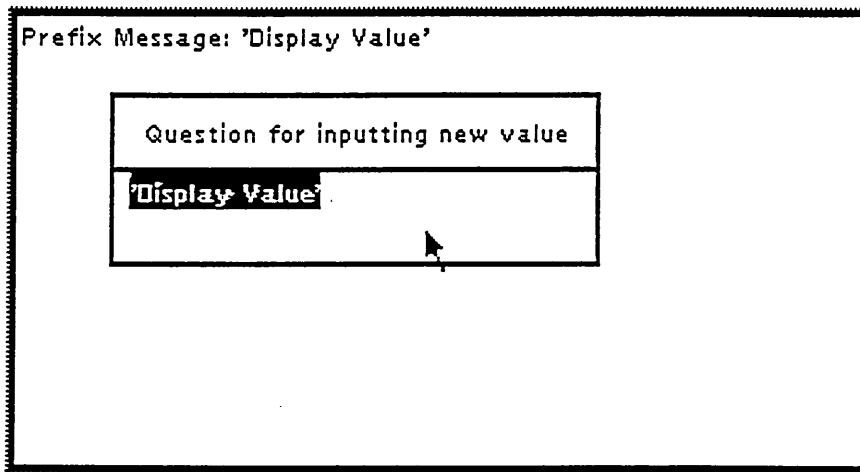
<question> A String which is used to prompt for new value.

<yBM> Used to determine Interaction menu, if any."

```

+#+ (label 'Title Msg' (Input Any String with (0 ) noMsgArgs ) "" "" )
  (displayValue 'Display Value' (Input Any String with (0 ) noMsgArgs ) ""Not Specified"" )
  (modelUpdate 'Updating Model' (Output Message with (1 ) noMsgArgs ) )
  (question 'String For Inputting New Value' (Input Any String with (0 ) noMsgArgs ) "" "" )
  (yBM 'YBM' (Input Any Menu with (0 ) noMsgArgs ) )

```



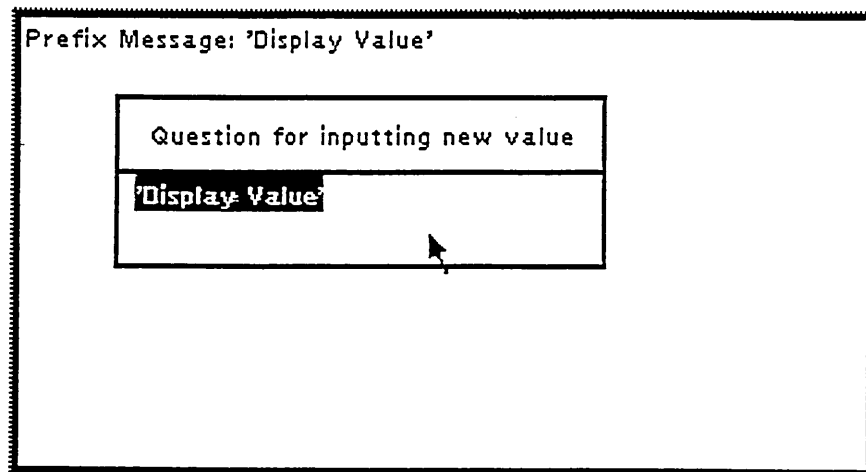
SlotDescription

*STRING VIEW AND REVISE IPVC

An IPVC which allows Strings to be displayed and modified. The value is displayed in a box, and may be prefixed with a specific message. The value may be modified by pressing the mouse button while over the box. A prompt is then given to enter the new value, and the model is informed of the new value using another linkage slot.

<label> Prefix displayed in front of value.
 <displayValue> String value to be displayed.
 <modelUpdate> Message used to inform model of new value.
 <question> A String which is used to prompt for new value.
 <yBM> Used to determine Interaction menu, if any."

```
↑#( (label 'Title Msg' (Input Any String with (0 ) noMsgArgs ) "" "" )
    (displayValue 'Display Value' (Input Any String with (0 ) noMsgArgs ) "" "Not Specified" )
    (modelUpdate 'Updating Model' (Output Message with (1 ) noMsgArgs ) )
    (question 'String For Inputting New Value' (Input Any String with (0 ) noMsgArgs ) "" "" )
    (yBM 'YBM' (Input Any Menu with (0 ) noMsgArgs ) ) )
```



SlotDescription

*GENERAL PURPOSE VIEW ONLY IPVC

A general purpose IPVC which allows any type of value to be displayed. The value displayed is determined using the printString method in conjunction with a linkage slot. The value is displayed in a box, and may be prefixed with a specific message.

<label> Prefix displayed in front of value.

<displayValue> Value to be displayed.

<yBM> Used to determine Interaction menu, if any."

```
↑#( (label 'Title Msg' (Input Any String with (0) noMsgArgs) "" "" )  
    (displayValue 'Display Value' (Input Message with (0) noMsgArgs) "" 'Not Specified' )  
    (yBM 'YBM' (Input Any Menu with (0) noMsgArgs)))
```

Prefix Message: 'Not Specified'

IPVCStringView class methodsFor: slot definitions

SlotDescription

"STRING VIEW ONLY IPVC

An IPVC which allows Strings to be displayed. The value is displayed in a box, and may be prefixed with a specific message

<label> Prefix displayed in front of value.

<displayValue> String value to be displayed.

<yBM> Used to determine Interaction menu, if any."

```
↑#( (label 'Title Msg' (Input Any String with (0 ) noMsgArgs ) "" "" )  
    (displayValue 'Display Value' (Input Any String with (0 ) noMsgArgs ) ""Not Specified"" )  
    (yBM 'YBM' (Input Any Menu with (0 ) noMsgArgs ) ) )
```

Prefix Message: 'Not Specified'

PPVCUserModelView class methodsFor: Slot definitions

SlotDescription

*Special Part PPVC - PPVCUSERMODEL

This is an example of of a Special Part PPVC which provides additional functions to those implemented by the UserModel Class. These functions can then be accessed by any PVCs attached to this PPVC.

The additional functions particularly enable an individual User Model method and class knowledge to be selected for inspection / modification.

The following Linkage slots are used to communicate with the User Model, and show the maximum number of slots supported by this implementation, i.e. 30"

```
↑#( (changeClassApplicationIncrease 'Change Class Application increase' (output Message with (2 ) noMsgArgs ) )
    (classApplicationIncrease 'Class Application increase' (input Message with (1 ) noMsgArgs ) )
    (changeClassUsage 'Change Class Usage' (output Message with (2 ) noMsgArgs ) )
    (classUsage 'Class Usage' (input Message with (1 ) noMsgArgs ) )
    (changeClassExpertiseLevel 'Change Class Expertise Level' (output Message with (2 ) noMsgArgs ) )
    (classExpertiseLevel 'Class Expertise Level' (input Message with (1 ) noMsgArgs ) )
    (changeClassLevel2Trigger 'Change Class Level2 Trigger' (output Message with (2 ) noMsgArgs ) )
    (classLevel2Trigger 'Class Level2 Trigger' (input Message with (1 ) noMsgArgs ) )
    (changeClassLevel3Trigger 'Change Class Level3 Trigger' (output Message with (2 ) noMsgArgs ) )
    (classLevel3Trigger 'Class Level3 Trigger' (input Message with (1 ) noMsgArgs ) )
    (classOverRideOff 'Class OverRide Off' (output Message with (1 ) noMsgArgs ) )
    (changeMethodClassIncrease 'Change Method Class Increase' (output Message with (3 ) noMsgArgs ) )
    (methodClassIncrease 'Method Class Increase' (input Message with (2 ) noMsgArgs ) )
    (changeMethodUsage 'Change Method Usage' (output Message with (3 ) noMsgArgs ) )
    (methodUsage 'Method Usage' (input Message with (2 ) noMsgArgs ) )
    (changeMethodExpertiseLevel 'Change Method Expertise Level' (output Message with (3 ) noMsgArgs ) )
    (methodExpertiseLevel 'Method Expertise Level' (input Message with (2 ) noMsgArgs ) )
    (changeMethodLevel2Trigger 'Change Method Level2 Trigger' (output Message with (3 ) noMsgArgs ) )
    (methodLevel2Trigger 'Method Level2 Trigger' (input Message with (2 ) noMsgArgs ) )
    (changeMethodLevel3Trigger 'Change Method Level3 Trigger' (output Message with (3 ) noMsgArgs ) )
    (methodLevel3Trigger 'Method Level3 Trigger' (input Message with (2 ) noMsgArgs ) )
    (methodOverRideOff 'Method OverRide Off' (output Message with (2 ) noMsgArgs ) )
    (availableMethods 'available methods' (input Message with (1 ) noMsgArgs ) )
    (errorWithClass 'Error with Class' (output Message with (1 ) noMsgArgs ) )
    (useClass 'Use of Class' (output Message with (1 ) noMsgArgs ) )
    (errorWithMethod 'Error with Method' (output Message with (2 ) noMsgArgs ) )
    (useMethod 'Use of Method' (output Message with (2 ) noMsgArgs ) )
    (removeClass 'Remove Class' (output Message with (1 ) noMsgArgs ) )
    (removeMethod 'Remove Method' (output Message with (2 ) noMsgArgs ) )
    (yBM 'yBM' (Input Any Menu with (0 ) noMsgArgs ) )
)
```

Appendix G

Class Hierarchy for Object Oriented User Interface Management System Implementation.

This appendix contains the Class Hierarchies for the proposed UIMS implementation. Due to its length, the actual Smalltalk 80 source code is omitted. This is available for inspection upon request from Sheffield City Polytechnic.

Contents :-

- (1) PVC Combined View and Separation Controller Class Hierarchy,
- (2) PVC Interaction Controller Class Hierarchy,
- (3) Extended Lean Cuisine Class Hierarchy.

Extended Lean Cuisine Implementation - Class Hierarchy :-

Object ()

Meneme ('title' 'owner')

Menu ('size' 'itemList' 'selected' 'selection' 'view' 'oneOn')

MCMenu ()

RealMCMenu ()

VirtualMCMenu ()

MEMenu ()

RealMEMenu ()

VirtualMEMenu ()

Terminator ()

Bistable ('onMsg' 'offMsg' 'state')

Monostable ('selectMsg'

Object Oriented User Interface Management System Implementation - Class Hierarchy for Interaction Controller Component :-

```
Object ()
  Controller ('model' 'view' 'sensor' )
    MouseMenuController ('redButtonMenu' 'redButtonMessages' 'yellowButtonMenu'
'yellowButtonMessages' 'blueButtonMenu' 'blueButtonMessages' )
      ScrollController ('scrollBar' 'marker' )
        PVCIC ('status' 'cursor' )
          IPVCListIC ()
          IPVCSliderIC ()
          IPVCStringIC ('paragraph' )
            IPVCStringIOIC ()
            IPVCTextEditorIC ('startBlock' 'stopBlock' 'beginTypeInBlock' 'emphasisHere' 'initialText'
'selectionShowing' )
              IPVCSwitchIC ()
              PPVCIC ()
              PPVCTopIC ('savedArea'
```

Object Oriented User Interface Management System Implementation - Class Hierarchy for Combined View/Separation Controller Components :-

```
Object ()
  View ('model' 'controller' 'superView' 'subViews' 'transformation' 'viewport' 'window' 'displayTransformation'
'insetDisplayBox' 'borderWidth' 'borderColor' 'insideColor' 'boundingBox' )
    PVCView ('links' 'slotValues' 'slotDefaults' 'slotArgs' 'inputSlotMsgs' 'outputSlotMsgs' 'changedSlots' )
      IPVCMultiView ('multiSlots' )
        IPVCBarChartView ()
      IPVCView ()
        IPVCGraphView ()
        IPVCHorizontalSliderView ('sliderKnobForm' 'underKnobForm' 'sliderTransformation'
'oldDTTransformation' 'currentCursorPos' )
          IPVCVerticalSliderView ()
        IPVCListView ('list' 'selection' 'topDelimiter' 'bottomDelimiter' 'lineSpacing' 'emphasisOn' 'itemList'
)
          IPVCStringIOView ()
            IPVCGeneralIOView ()
          IPVCStringView ()
            IPVCGeneralView ()
          IPVCSwitchView ('displayObject' )
            IPVCButtonView ()
            IPVCSwitchView2 ('switchState' )
          IPVCTextView ()
            IPVCStringEditorView ()
          PPVCView ('label' 'default' 'changed' 'partMsg' )
            PPVCTopView ('active' )
            SpecialPPVCView ()
              PPVCUserModelView ('currentClass' 'currentMethod'
```

Appendix H.

Example Interfaces and Associated Code Generated by the User Interface Management System.

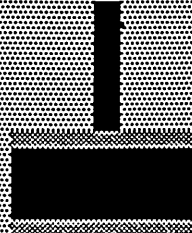
This appendix contains example direct manipulation user interfaces generated using the proposed UIMS tool-set. For each example, a screen dump of the actual working interface is given, and the underlying automatically generated PPVC description is listed.

Contents :-

- (1) Interface on a SparkPlug,
- (2) Interface on an Engine,
- (3) Interface on a DieselEngine,
- (4) Interface on a Tyre,
- (5) Interface on a Tyre,
- (6) Interface on a Chassis,
- (7) Interface on a Car,
- (8) Interface on a Car,
- (9) Interface on a Car,
- (10) Interface on a User Model,
- (11) Interface on a Dept,
- (12) Interface on a Person,
- (13) Interface on a Person,
- (14) Interface on a Detailed Person,
- (15) Interface on a Detailed Person,
- (16) Interface used for inspecting existing active PVC Slot values,
- (17) Interface used to Align existing active PVCs.

Bosch Special

Sp eff = 0.85



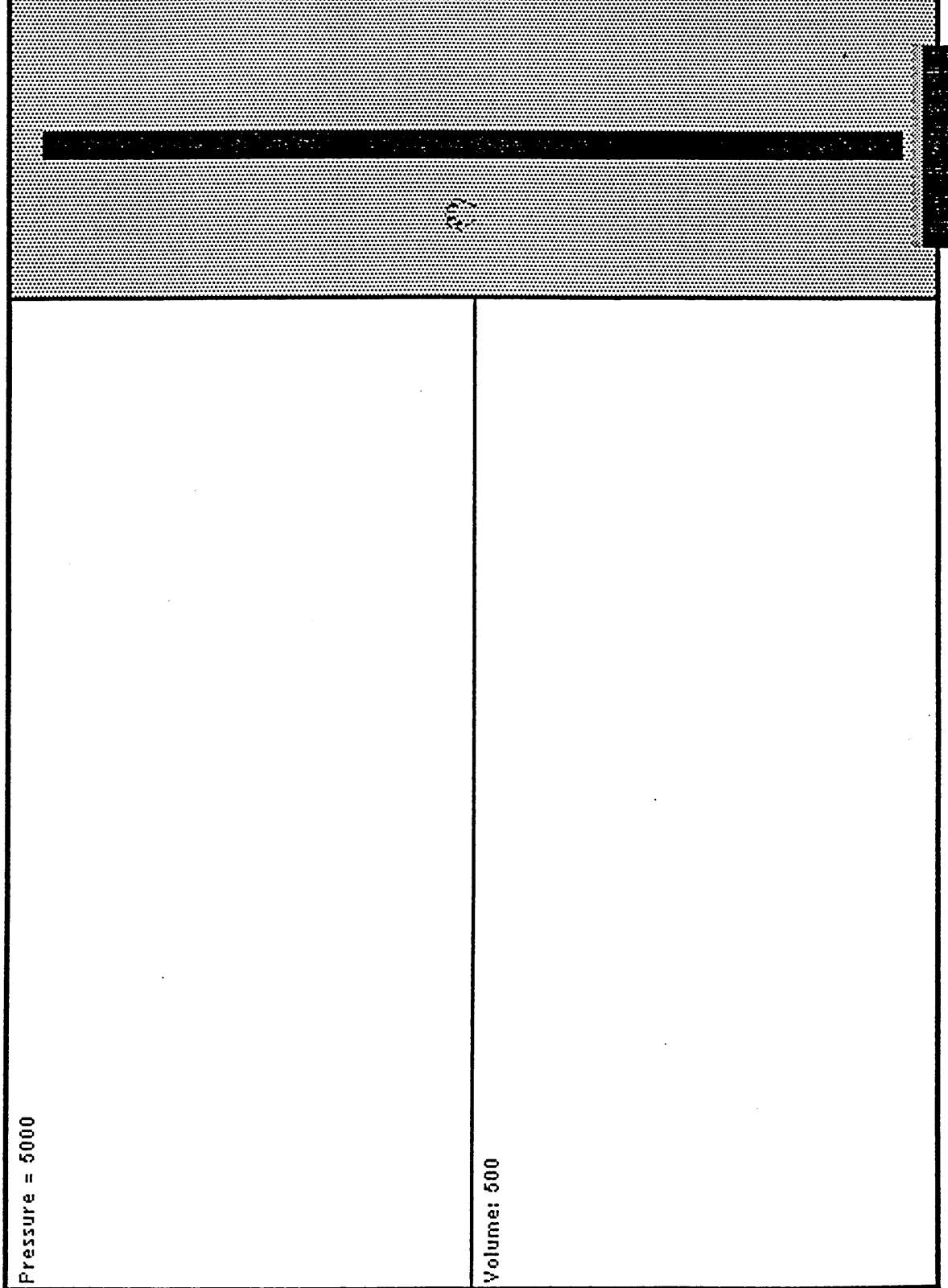
Interface 1

Default		
Sp eff = 0.85		
	Engine Type = 3	Power = 97.75
	SuperCharged Light Weight Version	Weight = 350
	----- SuperCharged Light Weight Version Default SuperCharged Version Cheapo Boring Efficient Engine -----	Size = 1999


Interface 2

Engine Type = 3	Power = 103.5
<div>-----</div> <div>Diesel Default</div> <div>Diesel SuperCharged Version</div> <div>Diesel Cheapo Boring Efficient Engine</div> <div>Diesel SuperCharged Light Weight Version</div> <div>-----</div>	Weight = 360
	Size = 1999
Diesel SuperCharged Light Weight Version	Efficiency = 0.9

Interface 3



Interface 4

<div style="background-color: #cccccc; padding: 5px;"> <div style="background-color: black; color: white; text-align: center; padding: 2px;"> [Redacted Title Bar] </div> </div>		<div style="text-align: center;">  </div>	<div style="text-align: center;"> Deflate </div>
<div style="text-align: center;"> Pressure = 5000 </div>	<div style="text-align: center;"> Volume: 500 </div>		

Interface 5

Chassis No: 6

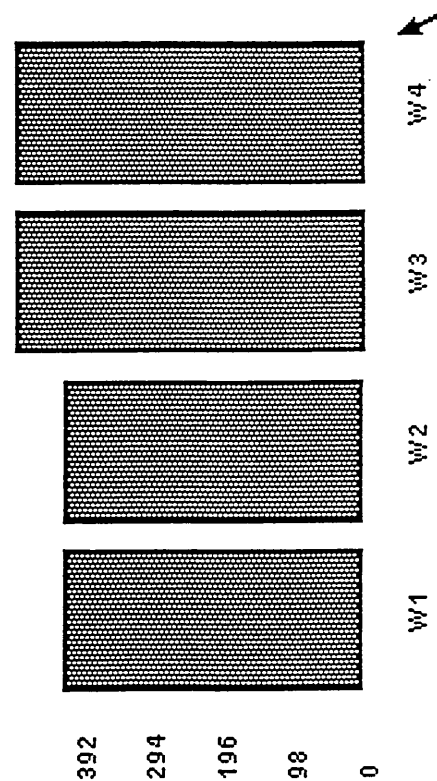
Frame C
Frame B
Frame D
Frame E
Frame F
Frame A
Default 2
Default 3
Default 4

deso: Frame F

Chassis wt: 130

Interface 6

Chassis No: 1	
Frame C Frame B Frame D Frame E Frame F Frame A Default 2 Default 3 Default 4 -----	desc: Frame A
	Chassis wt: 100
	Total W't: 1400
	Accel = 274.51

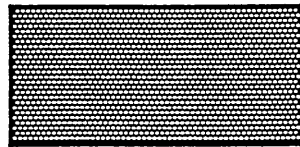
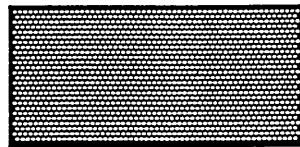
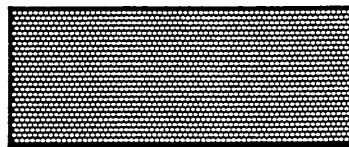
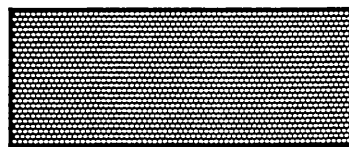


Interface 7

Default	
Sp eff = 0.85	
Engine Type = 1	Power = 51.0
Default	Weight = 300
SuperCharged Light Weight Version Default SuperCharged Version Cheapo Boring Efficient Engine -----	Size = 1300
<div style="display: flex; justify-content: space-around; align-items: flex-end;"> <div style="text-align: center;"> <p>392</p> </div> <div style="text-align: center;"> <p>294</p> </div> <div style="text-align: center;"> <p>196</p> </div> <div style="text-align: center;"> <p>98</p> </div> <div style="text-align: center;"> <p>0</p> </div> </div> <div style="display: flex; justify-content: space-around; margin-top: 10px;"> <p>W1</p> <p>W2</p> <p>W3</p> <p>W4</p> </div>	
Total Wt: 1400	
Accel = 274.51	

Interface 8

Interface 9

Pressure = 420		Pressure = 420		Inflate		Deflate	
Volume: 30		Volume: 30		Inflate		Deflate	
Pressure = 490		Pressure = 490		Inflate		Deflate	
Volume: 35		Volume: 35		Inflate		Deflate	
<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;">  W1 </div> <div style="text-align: center;">  W2 </div> <div style="text-align: center;">  W3 </div> <div style="text-align: center;">  W4 </div> </div>				<div style="display: flex; justify-content: space-between;"> <div> <p>392</p> <p>294</p> <p>196</p> <p>98</p> <p>0</p> </div> <div> <p>Total Wt: 1400</p> <p>Accel = 274.51</p> </div> </div>			

Model Title: First Test Model		[:(j sqrt) * 10]	
----- class3 class1 class2 -----			
----- method1 method2 -----			
App Err Count: 0		Default App Increase: 0.1	Default L2 Trigger: 35
		Default Class Increase: 0.1	Default L3 Trigger: 50
App Expertise: 1		Class Expertise: 1	Method Expertise: 1
App Usage: 0.74		Class Usage: 0.2	Method Usage: 1
App L2 Trigger: 35		Class L2 Trigger: 35	Method L2 Trigger: 35
App L3 Trigger: 50		Class L3 Trigger: 50	Method L3 Trigger: 50
App Last Used: 17 May 1990		Class App Increase: 0.1	Method Class Increase: 0.1
Application OverRide Off		Class OverRide Off	Method OverRide Off
Use Application		Use Class	Use Method
Error with Application		Error With Class	Error with Method

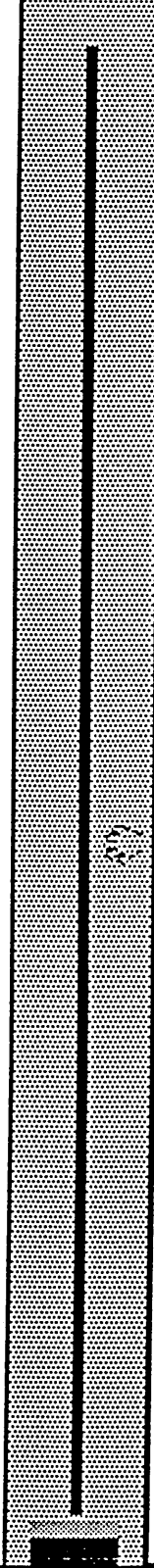
Interface 10

Dept Name = Physics	
Head = John	
10 Storey Block	
No Of Staff = 0	Take on Staff

Interface 11

Name = paul				
Not Known				
Date Of Birth = 17 May 1990				

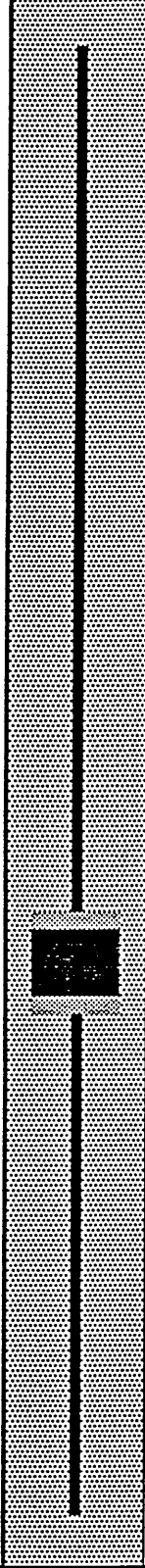
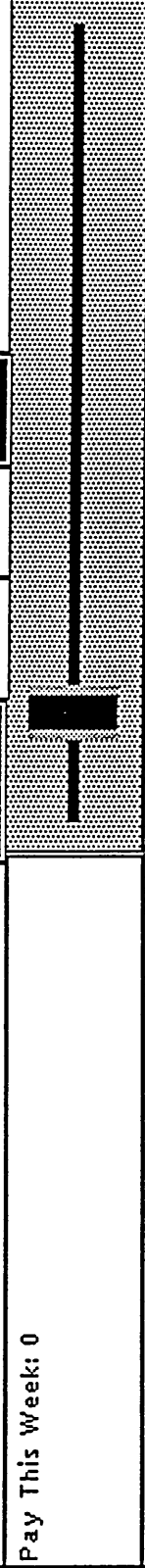
Interface 12

Name = paul	Not Known
Date Of Birth = 17 May 1990	
	
Dept Name = Computer Studies	
Head = paul	
Heriot House	
No Of Staff = 0	Take on Staff

Interface 13

Name = John	Not Known
Date Of Birth = 26 October 1944	
<div> <div></div> <div></div> </div>	
Salary: 5	Week End
Hours This Week: 0	Work an Hour
Yearly Pay: 0	Pay This Week: 0
<div> <div></div> <div></div> </div>	
Salary Increase: 31	Do Salary Increase

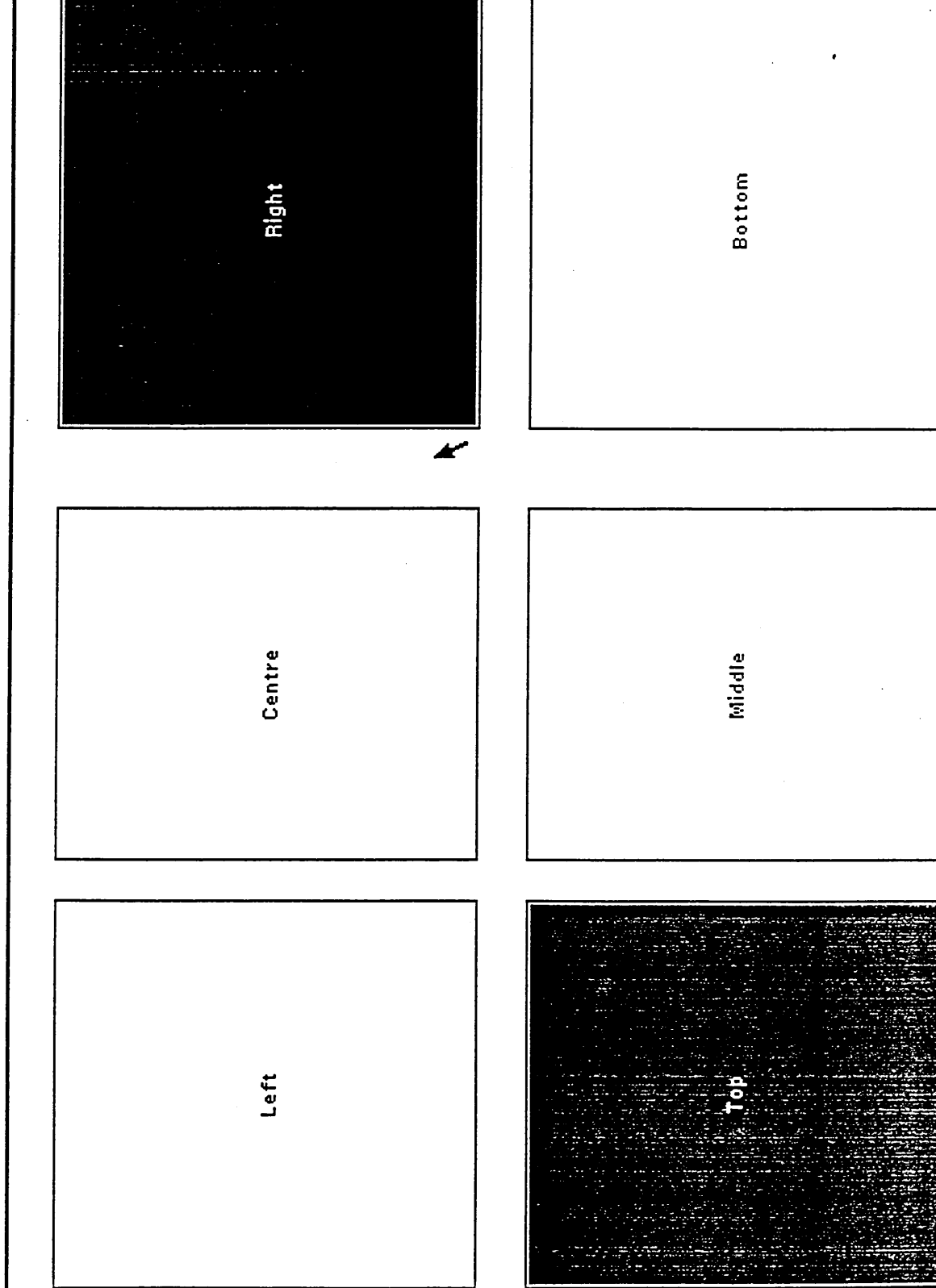
Interface 14

Name = john	Not Known			
Date Of Birth = 27 January 1952				
				
Salary: 5	Week End			
Hours This Week: 0	Work To Do = 10			
Yearly Pay: 0	SWITCH	1	5	10
Pay This Week: 0				
Salary Increase: 17	Do Salary Increase			

Interface 15

IPVC Type = an IPVCGeneralView	
<div> <div>-----</div> <div>label</div> <div>displayValue</div> <div>YBM</div> <div>-----</div> </div>	Slot Title = YBM
	Comms = Input
	Value = 'Not Specified'
	Value Class = No Value
	Change

Interface 16



Interface 17

Sparkplug methodsFor: PPVCs

sparkplugView1: aPPVC

"Example Interface 1"

aPPVC isNil

ifTrue: [↑self sparkplugView1 instantiation]

ifFalse:

[self sparkplugView1s1: aPPVC.

self sparkplugView1s2: aPPVC.

self sparkplugView1s3: aPPVC.

aPPVC add!PVCLinks: nil]

Sparkplug methodsFor: Sub PPVCs

sparkplugView1instantiation

```
↑((PPVCView new)
  slot: #yBM value: nil type: #Literal)
```

sparkplugView1s1: aPPVC

```
aPPVC
  addSubView: ((IPVCGeneralView new)
    slot: #label value: 'Sp eff =' type: #Literal;
    slot: #displayValue value: #efficiency type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.398 corner: 1.0@0.74 )
```

sparkplugView1s2: aPPVC

```
aPPVC
  addSubView: ((IPVCHorizontalSliderView new)
    slot: #sliderValue value: #efficiency type: #Message;
    slot: #sliderUpdate value: #efficiency type: #Message;
    slot: #sliderRange value: (0 to: 1 by: 0.05) type: #Literal;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.74 corner: 1.0@1.0 )
```

sparkplugView1s3: aPPVC

```
aPPVC
  addSubView: ((IPVCStringEditorView new)
    slot: #inputString value: #name type: #Message;
    slot: #modelUpdate value: #name type: #Message;
    slot: #yBM value: (Menu onString: 'RealMCMenu false Top[(MonoStable accept Accept) ]') type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.0 corner: 1.0@0.398 )
```

Engine methodsFor: PPVCs

engineView1: aPPVC

"Example Interface 2"

aPPVC isNil

ifTrue: [↑self engineView1instantiation]

ifFalse:

```
[self engineView1s1: aPPVC.  
self engineView1s2: aPPVC.  
self engineView1s3: aPPVC.  
self engineView1s4: aPPVC.  
self engineView1s5: aPPVC.  
self engineView1s6: aPPVC.  
self engineView1s7: aPPVC.  
aPPVC addIPVCLinks: nil]
```

Engine methodsFor: Sub PPVCs

engineView1 instantiation

```
↑((PPVCView new)
  slot: #yBM value: nil type: #Literal)
```

engineView1s1: aPPVC

```
aPPVC
  addPPVC: #sparkplugView1
    partMsg: #sparkplug
    variablePPVC: #variablePVC
    at: (0.0@0.0 corner: 1.0@0.404 )
```

engineView1s2: aPPVC

```
aPPVC
  addSubView: ((IPVCStringIOView new)
    slot: #label value: " type: #Literal;
    slot: #displayValue value: #description type: #Message;
    slot: #modelUpdate value: #engineDescription type: #Message;
    slot: #question value: 'engine description ?' type: #Literal;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.559 corner: 0.476@0.736 )
```

engineView1s3: aPPVC

```
aPPVC
  addSubView: ((IPVCListView new)
    slot: #currentItem value: #description type: #Message;
    slot: #newSelection value: #engineDescription type: #Message;
    slot: #itemList value: #availableDescriptions type: #Message;
    slot: #stringPrint value: false type: #Literal;
    slot: #oneItem value: false type: #Literal;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.736 corner: 0.476@1.0 )
```

engineView1s4: aPPVC

```
aPPVC
  addSubView: ((IPVCGeneralView new)
    slot: #label value: 'Power = ' type: #Literal;
    slot: #displayValue value: #power type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.476@0.404 corner: 1.0@0.601 )
```

engineView1s5: aPPVC

```
aPPVC
  addSubView: ((IPVCGeneralView new)
    slot: #label value: 'Weight = ' type: #Literal;
    slot: #displayValue value: #weight type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.476@0.601 corner: 1.0@0.803 )
```

engineView1s6: aPPVC

```
aPPVC
  addSubView: ((IPVCGeneralIOView new)
    slot: #label value: 'Engine Type = ' type: #Literal;
    slot: #displayValue value: #engineType type: #Message;
    slot: #modelUpdate value: #engineType type: #Message;
    slot: #question value: 'new Type ?' type: #Literal;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.00606@0.403 corner: 0.476@0.56 )
```

engineView1s7: aPPVC

```
aPPVC
  addSubView: ((IPVCGeneralView new)
    slot: #label value: 'Size = ' type: #Literal;
    slot: #displayValue value: #size type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.476@0.803 corner: 1.0@1.0 )
```


DieselEngine methodsFor: PPVCs

engineView1: aPPVC

"Example Interface 3"

aPPVC isNil

ifTrue: [↑self engineView1 instantiation]

ifFalse:

[self engineView1s1: aPPVC.
self engineView1s2: aPPVC.
self engineView1s3: aPPVC.
self engineView1s4: aPPVC.
self engineView1s5: aPPVC.
self engineView1s6: aPPVC.
self engineView1s7: aPPVC.
aPPVC addIPVCLinks: nil]

DieselEngine methodsFor: Sub PPVCs

engineView1instantiation

```
↑((PPVCView new)
    slot: #yBM value: nil type: #Literal)
```

engineView1s1: aPPVC

```
aPPVC
    addSubView: ((IPVCStringIOView new)
        slot: #label value: " type: #Literal;
        slot: #displayValue value: #description type: #Message;
        slot: #modelUpdate value: #engineDescription type: #Message;
        slot: #question value: 'engine description ?' type: #Literal;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.00362@0.717 corner: 0.476@0.998 )
```

engineView1s2: aPPVC

```
aPPVC
    addSubView: ((IPVCListView new)
        slot: #currentItem value: #description type: #Message;
        slot: #newSelection value: #engineDescription type: #Message;
        slot: #itemList value: #availableDescriptions type: #Message;
        slot: #stringPrint value: false type: #Literal;
        slot: #oneItem value: false type: #Literal;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.00362@0.233 corner: 0.476@0.717 )
```

engineView1s3: aPPVC

```
aPPVC
    addSubView: ((IPVCGeneralView new)
        slot: #label value: 'Power = ' type: #Literal;
        slot: #displayValue value: #power type: #Message;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.476@0.0 corner: 1.0@0.233 )
```

engineView1s4: aPPVC

```
aPPVC
    addSubView: ((IPVCGeneralView new)
        slot: #label value: 'Weight = ' type: #Literal;
        slot: #displayValue value: #weight type: #Message;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.476@0.233 corner: 1.0@0.552 )
```

engineView1s5: aPPVC

```
aPPVC
    addSubView: ((IPVCGeneralIOView new)
        slot: #label value: 'Engine Type = ' type: #Literal;
        slot: #displayValue value: #engineType type: #Message;
        slot: #modelUpdate value: #engineType type: #Message;
        slot: #question value: 'new Type ?' type: #Literal;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.0@0.0 corner: 0.476@0.233 )
```

engineView1s6: aPPVC

```
aPPVC
    addSubView: ((IPVCGeneralView new)
        slot: #label value: 'Size = ' type: #Literal;
        slot: #displayValue value: #size type: #Message;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.476@0.552 corner: 1.0@0.784 )
```

engineView1s7: aPPVC

```
aPPVC
    addSubView: ((IPVCGeneralView new)
        slot: #label value: 'Efficiency = ' type: #Literal;
        slot: #displayValue value: #efficiency type: #Message;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.476@0.784 corner: 1.0@1.0 )
```

Tyre methodsFor: PPVCs

Tyre1: aPPVC

"Example Interface 4"

```
aPPVC isNil  
  ifTrue: [↑self Tyre1instantiation]  
  ifFalse:  
    [self Tyre1s1: aPPVC.  
     self Tyre1s2: aPPVC.  
     self Tyre1s3: aPPVC.  
     aPPVC addIPVCLinks: nil]
```

Tyre2: aPPVC

"Example Interface 5"

```
aPPVC isNil  
  ifTrue: [↑self Tyre2instantiation]  
  ifFalse:  
    [self Tyre2s1: aPPVC.  
     self Tyre2s2: aPPVC.  
     self Tyre2s3: aPPVC.  
     self Tyre2s4: aPPVC.  
     self Tyre2s5: aPPVC.  
     aPPVC addIPVCLinks: nil]
```

Type methodsFor: Sub PPVCs

Tyre1instantiation

```
↑((PPVCView new)
    slot: #yBM value: nil type: #Literal)
```

Tyre1s1: aPPVC

```
aPPVC
    addSubView: ((IPVCGeneralView new)
        slot: #label value: 'Pressure = ' type: #Literal;
        slot: #displayValue value: #pressure type: #Message;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.0@0.0 corner: 0.763@0.502 )
```

Tyre1s2: aPPVC

```
aPPVC
    addSubView: ((IPVCGeneralIOView new)
        slot: #label value: 'Volume: ' type: #Literal;
        slot: #displayValue value: #volume type: #Message;
        slot: #modelUpdate value: #volume type: #Message;
        slot: #question value: 'New Volume' type: #Literal;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.0@0.502 corner: 0.763@1.0 )
```

Tyre1s3: aPPVC

```
aPPVC
    addSubView: ((IPVCVerticalSliderView new)
        slot: #sliderValue value: #volume type: #Message;
        slot: #sliderUpdate value: #volume type: #Message;
        slot: #sliderRange value: (1 to: 500 by: 10) type: #Literal;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.763@0.0 corner: 1.0@1.0 )
```

Tyre2instantiation

```
↑((PPVCView new)
    slot: #yBM value: nil type: #Literal)
```

Tyre2s1: aPPVC

```
aPPVC
    addSubView: ((IPVCGeneralView new)
        slot: #label value: 'Pressure = ' type: #Literal;
        slot: #displayValue value: #pressure type: #Message;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.0@0.0 corner: 0.763@0.502 )
```

Tyre2s2: aPPVC

```
aPPVC
    addSubView: ((IPVCGeneralIOView new)
        slot: #label value: 'Volume: ' type: #Literal;
        slot: #displayValue value: #volume type: #Message;
        slot: #modelUpdate value: #volume type: #Message;
        slot: #question value: 'New Volume' type: #Literal;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.00483@0.502 corner: 0.763@0.815 )
```

Tyre2s3: aPPVC

```
aPPVC
    addSubView: ((IPVCVerticalSliderView new)
        slot: #sliderValue value: #volume type: #Message;
        slot: #sliderUpdate value: #volume type: #Message;
        slot: #sliderRange value: (1 to: 500 by: 10) type: #Literal;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.763@0.0 corner: 1.0@0.815 )
```

Tyre2s4: aPPVC

```
aPPVC
    addSubView: ((IPVCButtonView new)
        slot: #label value: 'Inflate' type: #Literal;
        slot: #switchPress value: #inflate type: #Message;
        slotArgs: #switchPress value: #(10 );
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.0@0.815 corner: 0.502@1.0 )
```

Tyre2s5: aPPVC

aPPVC

```
addSubview: ((IPVCButtonView new)
  slot: #label value: 'Deflate' type: #Literal;
  slot: #switchPress value: #inflate type: #Message;
  slotArgs: #switchPress value: #(-10 );
  slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.502@0.815 corner: 1.0@1.0 )
```

Chassis methodsFor: PPVCs

chassis1: aPPVC

"Example Interface 6"

aPPVC isNil

ifTrue: [↑self chassis1instantiation]

ifFalse:

[self chassis1s1: aPPVC.

self chassis1s2: aPPVC.

self chassis1s3: aPPVC.

self chassis1s4: aPPVC.

aPPVC addIPVCLinks: nil]

Chassis methodsFor: Sub PPVCs

chassis1instantiation

```
↑((PPVCView new)
  slot: #yBM value: nil type: #Literal)
```

chassis1s1: aPPVC

```
aPPVC
  addSubview: ((IPVCGeneralIOView new)
    slot: #label value: 'Chassis No: ' type: #Literal;
    slot: #displayValue value: #chassisNumber type: #Message;
    slot: #modelUpdate value: #chassisNumber type: #Message;
    slot: #question value: 'Type new Chassis Number' type: #Literal;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.0 corner: 1.0@0.502 )
```

chassis1s2: aPPVC

```
aPPVC
  addSubview: ((IPVCListView new)
    slot: #currentItem value: #description type: #Message;
    slot: #newSelection value: #chassisDescription type: #Message;
    slot: #itemList value: #availableDescriptions type: #Message;
    slot: #stringPrint value: false type: #Literal;
    slot: #oneItem value: false type: #Literal;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.502 corner: 0.5@1.0 )
```

chassis1s3: aPPVC

```
aPPVC
  addSubview: ((IPVCStringView new)
    slot: #label value: 'desc: ' type: #Literal;
    slot: #displayValue value: #description type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.5@0.502 corner: 1.0@0.776 )
```

chassis1s4: aPPVC

```
aPPVC
  addSubview: ((IPVCGeneralView new)
    slot: #label value: 'Chassis wt: ' type: #Literal;
    slot: #displayValue value: #weight type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.5@0.776 corner: 1.0@1.0 )
```

Car methodsFor: PPVCs

Car1: aPPVC

"Example Interface 7"

aPPVC isNil

ifTrue: [↑self Car1instantiation]

ifFalse:

```
[self Car1s1: aPPVC.  
self Car1s2: aPPVC.  
self Car1s3: aPPVC.  
self Car1s4: aPPVC.  
aPPVC addIPVCLinks: nil]
```

Car2: aPPVC

"Example Interface 8"

aPPVC isNil

ifTrue: [↑self Car2instantiation]

ifFalse:

```
[self Car2s1: aPPVC.  
self Car2s2: aPPVC.  
self Car2s3: aPPVC.  
self Car2s4: aPPVC.  
aPPVC addIPVCLinks: nil]
```

Car3: aPPVC

"Example Interface 9"

aPPVC isNil

ifTrue: [↑self Car3instantiation]

ifFalse:

```
[self Car3s1: aPPVC.  
self Car3s2: aPPVC.  
self Car3s3: aPPVC.  
self Car3s4: aPPVC.  
self Car3s5: aPPVC.  
self Car3s6: aPPVC.  
self Car3s7: aPPVC.  
aPPVC addIPVCLinks: nil]
```


Car methodsFor: Sub PPVCs

Car1instantiation

```
↑((PPVCView new)
    slot: #yBM value: nil type: #Literal)
```

Car1s1: aPPVC

```
aPPVC
    addPPVC: #chassis1
        partMsg: #chassis
        variablePPVC: #variablePVC
        at: (0.0@0.0 corner: 1.0@0.574 )
```

Car1s2: aPPVC

```
aPPVC
    addSubView: ((IPVCBarChartView new)
        slot: #label value: 'No Label' type: #Literal;
        multiSlot: #barValues
        values: ((OrderedCollection new) add: 'wheel1.pressure' asSymbol; add: 'wheel2.pressure' asSymbol; add:
'wheel3.pressure' asSymbol; add: 'wheel4.pressure' asSymbol; yourself)
        types: #(Message Message Message Message );
        multiSlot: #barTitles
        values: ((OrderedCollection new) add: 'W1'; add: 'W2'; add: 'W3'; add: 'W4'; yourself)
        types: #Literal;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.0@0.574 corner: 0.503@1.0 )
```

Car1s3: aPPVC

```
aPPVC
    addSubView: ((IPVCGeneralIOView new)
        slot: #label value: 'Total Wt: ' type: #Literal;
        slot: #displayValue value: #totalWeight type: #Message;
        slot: #modelUpdate value: #weight type: #Message;
        slot: #question value: 'New Car Weight' type: #Literal;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.503@0.574 corner: 1.0@0.776 )
```

Car1s4: aPPVC

```
aPPVC
    addSubView: ((IPVCGeneralView new)
        slot: #label value: 'Accel = ' type: #Literal;
        slot: #displayValue value: #acceleration type: #Message;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.503@0.776 corner: 1.0@1.0 )
```

Car2instantiation

```
↑((PPVCView new)
    slot: #yBM value: nil type: #Literal)
```

Car2s1: aPPVC

```
aPPVC
    addSubView: ((IPVCBarChartView new)
        slot: #label value: 'No Label' type: #Literal;
        multiSlot: #barValues
        values: ((OrderedCollection new) add: 'wheel1.pressure' asSymbol; add: 'wheel2.pressure' asSymbol; add:
'wheel3.pressure' asSymbol; add: 'wheel4.pressure' asSymbol; yourself)
        types: #(Message Message Message Message );
        multiSlot: #barTitles
        values: ((OrderedCollection new) add: 'W1'; add: 'W2'; add: 'W3'; add: 'W4'; yourself)
        types: #Literal;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.0@0.574 corner: 0.503@1.0 )
```

Car2s2: aPPVC

```
aPPVC
    addSubView: ((IPVCGeneralIOView new)
        slot: #label value: 'Total Wt: ' type: #Literal;
        slot: #displayValue value: #totalWeight type: #Message;
        slot: #modelUpdate value: #weight type: #Message;
        slot: #question value: 'New Car Weight' type: #Literal;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.503@0.574 corner: 1.0@0.776 )
```

Car2s3: aPPVC

```

aPPVC
  addSubview: ((IPVCGeneralView new)
    slot: #label value: 'Accel = ' type: #Literal;
    slot: #displayValue value: #acceleration type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.503@0.776 corner: 1.0@1.0 )

Car2s4: aPPVC
  aPPVC
    addPPVC: #engineView1
      partMsg: #engine
      variablePPVC: #variablePVC
      at: (0.0@0.0 corner: 1.0@0.574 )

Car3Instantiation
  ↑((PPVCView new)
    slot: #yBM value: nil type: #Literal)

Car3s1: aPPVC
  aPPVC
    addSubview: ((IPVCBarChartView new)
      slot: #label value: 'No Label' type: #Literal;
      multiSlot: #barValues
        values: ((OrderedCollection new) add: 'wheel1.pressure' asSymbol; add: 'wheel2.pressure' asSymbol; add:
'wheel3.pressure' asSymbol; add: 'wheel4.pressure' asSymbol; yourself)
        types: #(Message Message Message Message );
      multiSlot: #barTitles
        values: ((OrderedCollection new) add: 'W1'; add: 'W2'; add: 'W3'; add: 'W4'; yourself)
        types: #Literal;
      slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.0@0.574 corner: 0.503@1.0 )

Car3s2: aPPVC
  aPPVC
    addSubview: ((IPVCGeneralIOView new)
      slot: #label value: 'Total Wt: ' type: #Literal;
      slot: #displayValue value: #totalWeight type: #Message;
      slot: #modelUpdate value: #weight: type: #Message;
      slot: #question value: 'New Car Weight' type: #Literal;
      slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.503@0.574 corner: 1.0@0.776 )

Car3s3: aPPVC
  aPPVC
    addSubview: ((IPVCGeneralView new)
      slot: #label value: 'Accel = ' type: #Literal;
      slot: #displayValue value: #acceleration type: #Message;
      slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.503@0.776 corner: 1.0@1.0 )

Car3s4: aPPVC
  aPPVC
    addPPVC: #Tyre1
      partMsg: #wheel1
      variablePPVC: #variablePVC
      at: (0.0@0.0 corner: 0.507@0.289 )

Car3s5: aPPVC
  aPPVC
    addPPVC: 'Tyre1'
      partMsg: #wheel2
      variablePPVC: #variablePVC
      at: (0.507@0.00295 corner: 1.0@0.289 )

Car3s6: aPPVC
  aPPVC
    addPPVC: #Tyre2
      partMsg: #wheel3
      variablePPVC: #variablePVC
      at: (0.0@0.289 corner: 0.509@0.575 )

Car3s7: aPPVC
  aPPVC
    addPPVC: 'Tyre2'
      partMsg: #wheel4
      variablePPVC: #variablePVC

```

UserModel methodsFor: PPVCs

UserModelView1: aPPVC

"Example Interface 10"

aPPVC isNil

ifTrue: [↑self UserModelView1instantiation]

ifFalse:

```
[self UserModelView1s1: aPPVC.  
self UserModelView1s2: aPPVC.  
self UserModelView1s3: aPPVC.  
self UserModelView1s4: aPPVC.  
self UserModelView1s5: aPPVC.  
self UserModelView1s6: aPPVC.  
self UserModelView1s7: aPPVC.  
self UserModelView1s8: aPPVC.  
self UserModelView1s9: aPPVC.  
self UserModelView1s10: aPPVC.  
self UserModelView1s11: aPPVC.  
self UserModelView1s12: aPPVC.  
self UserModelView1s13: aPPVC.  
self UserModelView1s14: aPPVC.  
self UserModelView1s15: aPPVC.  
self UserModelView1s16: aPPVC.  
self UserModelView1s17: aPPVC.  
self UserModelView1s18: aPPVC.  
self UserModelView1s19: aPPVC.  
self UserModelView1s20: aPPVC.  
self UserModelView1s21: aPPVC.  
self UserModelView1s22: aPPVC.  
self UserModelView1s23: aPPVC.  
self UserModelView1s24: aPPVC.  
self UserModelView1s25: aPPVC.  
self UserModelView1s26: aPPVC.  
self UserModelView1s27: aPPVC.  
self UserModelView1s28: aPPVC.  
self UserModelView1s29: aPPVC.  
self UserModelView1s30: aPPVC.  
self UserModelView1s31: aPPVC.  
self UserModelView1s32: aPPVC.  
self UserModelView1s33: aPPVC.  
self UserModelView1s34: aPPVC.  
aPPVC addIPVCLinks: nil]
```

UserModel methodsFor: Sub PPVCs

UserModelView1Instantiation

```
↑((PPVCUserModelView new)
  slot: #changeClassApplicationIncrease value: #changeClass:applicationIncrease: type: #Message;
  slot: #classApplicationIncrease value: #classApplicationIncrease: type: #Message;
  slot: #changeClassUsage value: #changeClass:usage: type: #Message;
  slot: #classUsage value: #classUsage: type: #Message;
  slot: #changeClassExpertiseLevel value: #changeClass:expertiseLevel: type: #Message;
  slot: #classExpertiseLevel value: #classExpertiseLevel: type: #Message;
  slot: #changeClassLevel2Trigger value: #changeClass:level2Trigger: type: #Message;
  slot: #classLevel2Trigger value: #classLevel2Trigger: type: #Message;
  slot: #changeClassLevel3Trigger value: #changeClass:level3Trigger: type: #Message;
  slot: #classLevel3Trigger value: #classLevel3Trigger: type: #Message;
  slot: #classOverRideOff value: #changeClassOverRideOff: type: #Message;
  slot: #changeMethodClassIncrease value: #changeMethod:forClass:classIncrease: type: #Message;
  slot: #methodClassIncrease value: #methodClassIncrease:forClass: type: #Message;
  slot: #changeMethodUsage value: #changeMethod:forClass:usage: type: #Message;
  slot: #methodUsage value: #methodUsage:forClass: type: #Message;
  slot: #changeMethodExpertiseLevel value: #changeMethod:forClass:expertiseLevel: type: #Message;
  slot: #methodExpertiseLevel value: #methodExpertiseLevel:forClass: type: #Message;
  slot: #changeMethodLevel2Trigger value: #changeMethod:forClass:level2Trigger: type: #Message;
  slot: #methodLevel2Trigger value: #methodLevel2Trigger:forClass: type: #Message;
  slot: #changeMethodLevel3Trigger value: #changeMethod:forClass:level3Trigger: type: #Message;
  slot: #methodLevel3Trigger value: #methodLevel3Trigger:forClass: type: #Message;
  slot: #methodOverRideOff value: #changeMethodOverRideOff:forClass: type: #Message;
  slot: #availableMethods value: #methodsForClass: type: #Message;
  slot: #errorWithClass value: #errorWithClass: type: #Message;
  slot: #useClass value: #useClass: type: #Message;
  slot: #errorWithMethod value: #errorWithMethod:forClass: type: #Message;
  slot: #useMethod value: #useMethod:forClass: type: #Message;
  slot: #removeClass value: #removeClass: type: #Message;
  slot: #removeMethod value: #removeMethod:forClass: type: #Message;
  slot: #yBM value: nil type: #Literal)
```

UserModelView1s10: aPPVC

```
aPPVC
  addSubView: ((IPVCButtonView new)
    slot: #label value: 'Error with Application' type: #Literal;
    slot: #switchPress value: #errorWithApplication type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.941 corner: 0.351@1.0 )
```

UserModelView1s11: aPPVC

```
aPPVC
  addSubView: ((IPVCButtonView new)
    slot: #label value: 'Error With Class' type: #Literal;
    slot: #switchPress value: #errorWithCurrentClass type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.351@0.941 corner: 0.683@1.0 )
```

UserModelView1s12: aPPVC

```
aPPVC
  addSubView: ((IPVCButtonView new)
    slot: #label value: 'Error with Method' type: #Literal;
    slot: #switchPress value: #errorWithCurrentMethod type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.683@0.941 corner: 1.0@1.0 )
```

UserModelView1s13: aPPVC

```
aPPVC
  addSubView: ((IPVCGeneralOView new)
    slot: #label value: 'App Expertise:' type: #Literal;
    slot: #displayValue value: #applicationExpertiseLevel type: #Message;
    slot: #modelUpdate value: #applicationExpertiseLevel: type: #Message;
    slot: #question value: 'New Application Expertise Level (1 - 3)' type: #Literal;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.00346@0.486 corner: 0.351@0.545 )
```

UserModelView1s14: aPPVC

```
aPPVC
  addSubView: ((IPVCGeneralOView new)
    slot: #label value: 'App Err Count:' type: #Literal;
    slot: #displayValue value: #applicationErrorCount type: #Message;
    slot: #modelUpdate value: #applicationErrorCount: type: #Message;
```

slot: #question value: 'New Application Error Count' type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.0@0.383 corner: 0.351@0.486)

UserModelView1s15: aPPVC

aPPVC

addSubview: ((IPVCGeneralIOView new)
slot: #label value: 'App L2 Trigger: ' type: #Literal;
slot: #displayValue value: #applicationLevel2Trigger type: #Message;
slot: #modelUpdate value: #applicationLevel2Trigger type: #Message;
slot: #question value: 'New Application Level 2 Trigger' type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.0@0.615 corner: 0.351@0.686)

UserModelView1s16: aPPVC

aPPVC

addSubview: ((IPVCGeneralIOView new)
slot: #label value: 'App L3 Trigger: ' type: #Literal;
slot: #displayValue value: #applicationLevel3Trigger type: #Message;
slot: #modelUpdate value: #applicationLevel3Trigger type: #Message;
slot: #question value: 'New Application Level 3 Trigger' type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.00346@0.686 corner: 0.351@0.756)

UserModelView1s17: aPPVC

aPPVC

addSubview: ((IPVCGeneralView new)
slot: #label value: 'App Last Used: ' type: #Literal;
slot: #displayValue value: #applicationLastUsed type: #Message;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.00346@0.756 corner: 0.351@0.825)

UserModelView1s18: aPPVC

aPPVC

addSubview: ((IPVCGeneralIOView new)
slot: #label value: 'Class Expertise: ' type: #Literal;
slot: #displayValue value: #currentClassExpertiseLevel type: #Message;
slot: #modelUpdate value: #changeCurrentClassExpertiseLevel type: #Message;
slot: #question value: 'New Class Expertise Level (1-3)' type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.351@0.486 corner: 0.683@0.545)

UserModelView1s19: aPPVC

aPPVC

addSubview: ((IPVCGeneralIOView new)
slot: #label value: 'Method Expertise: ' type: #Literal;
slot: #displayValue value: #currentMethodExpertiseLevel type: #Message;
slot: #modelUpdate value: #changeCurrentMethodExpertiseLevel type: #Message;
slot: #question value: 'New Method Expertise Level (1-3)' type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.683@0.486 corner: 1.0@0.545)

UserModelView1s1: aPPVC

aPPVC

addSubview: ((IPVCStringIOView new)
slot: #label value: 'Model Title: ' type: #Literal;
slot: #displayValue value: #title type: #Message;
slot: #modelUpdate value: #title type: #Message;
slot: #question value: 'New Model Title' type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.0@0.0 corner: 0.428@0.0683)

UserModelView1s20: aPPVC

aPPVC

addSubview: ((IPVCGeneralIOView new)
slot: #label value: 'Class L2 Trigger: ' type: #Literal;
slot: #displayValue value: #currentClassLevel2Trigger type: #Message;
slot: #modelUpdate value: #changeCurrentClassLevel2Trigger type: #Message;
slot: #question value: 'New Class Level 2 Trigger' type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.351@0.615 corner: 0.683@0.686)

UserModelView1s21: aPPVC

aPPVC

```

addSubview: ((IPVCGeneralIOView new)
    slot: #label value: 'Class L3 Trigger:' type: #Literal;
    slot: #displayValue value: #currentClassLevel3Trigger type: #Message;
    slot: #modelUpdate value: #changeCurrentClassLevel3Trigger type: #Message;
    slot: #question value: 'New Class Level 3 Trigger' type: #Literal;
    slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.351@0.686 corner: 0.683@0.756 )

```

UserModelView1s22: aPPVC

aPPVC

```

addSubview: ((IPVCGeneralIOView new)
    slot: #label value: 'Method L2 Trigger:' type: #Literal;
    slot: #displayValue value: #currentMethodLevel2Trigger type: #Message;
    slot: #modelUpdate value: #changeCurrentMethodLevel2Trigger type: #Message;
    slot: #question value: 'New Method Level 2 Trigger' type: #Literal;
    slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.683@0.615 corner: 1.0@0.686 )

```

UserModelView1s23: aPPVC

aPPVC

```

addSubview: ((IPVCGeneralIOView new)
    slot: #label value: 'Method L3 Trigger:' type: #Literal;
    slot: #displayValue value: #currentMethodLevel3Trigger type: #Message;
    slot: #modelUpdate value: #changeCurrentMethodLevel3Trigger type: #Message;
    slot: #question value: 'New Method Level 3 Trigger' type: #Literal;
    slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.683@0.686 corner: 1.0@0.756 )

```

UserModelView1s24: aPPVC

aPPVC

```

addSubview: ((IPVCGeneralIOView new)
    slot: #label value: 'Method Class Increase:' type: #Literal;
    slot: #displayValue value: #currentMethodClassIncrease type: #Message;
    slot: #modelUpdate value: #changeCurrentMethodClassIncrease type: #Message;
    slot: #question value: 'New Method Class Increase' type: #Literal;
    slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.683@0.756 corner: 1.0@0.825 )

```

UserModelView1s25: aPPVC

aPPVC

```

addSubview: ((IPVCGeneralIOView new)
    slot: #label value: 'Class App Increase:' type: #Literal;
    slot: #displayValue value: #currentClassApplicationIncrease type: #Message;
    slot: #modelUpdate value: #changeCurrentClassApplicationIncrease type: #Message;
    slot: #question value: 'New Class Application Increase' type: #Literal;
    slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.351@0.756 corner: 0.683@0.825 )

```

UserModelView1s26: aPPVC

aPPVC

```

addSubview: ((IPVCButtonView new)
    slot: #label value: 'Application OverRide Off' type: #Literal;
    slot: #switchPress value: #applicationOverRideOff type: #Message;
    slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.00346@0.825 corner: 0.351@0.886 )

```

UserModelView1s27: aPPVC

aPPVC

```

addSubview: ((IPVCButtonView new)
    slot: #label value: 'Class OverRide Off' type: #Literal;
    slot: #switchPress value: #changeCurrentClassOverRideOff type: #Message;
    slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.351@0.825 corner: 0.683@0.886 )

```

UserModelView1s28: aPPVC

aPPVC

```

addSubview: ((IPVCButtonView new)
    slot: #label value: 'Method OverRide Off' type: #Literal;
    slot: #switchPress value: #changeCurrentMethodOverRideOff type: #Message;
    slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)

```

viewport: (0.683@0.825 corner: 0.998@0.886)

UserModelView1s29: aPPVC

aPPVC

addSubview: ((IPVCGraphView new)

slot: #curveBlock value: #learnFormula type: #Message;
slot: #minX value: 1 type: #Literal;
slot: #maxX value: 100 type: #Literal;
slot: #minY value: 1 type: #Literal;
slot: #maxY value: 100 type: #Literal;
slot: #step value: 1 type: #Literal;
slot: #yBM value: nil type: #Literal)

window: (0 @ 0 extent: 1 @ 1)

viewport: (0.427@0.0 corner: 1.0@0.32)

UserModelView1s2: aPPVC

aPPVC

addSubview: ((IPVCListView new)

slot: #currentItem value: #currentClass type: #Message;
slot: #newSelection value: #currentClass type: #Message;
slot: #itemList value: #availableClasses type: #Message;
slot: #stringPrint value: false type: #Literal;
slot: #oneItem value: false type: #Literal;
slot: #yBM value: (Menu onString: 'RealMEMenu false top[(MonoStable addClass Add New Class) (MonoStable

removeCurrentClass Remove Class)]) type: #Literal)

window: (0 @ 0 extent: 1 @ 1)

viewport: (0.0@0.0683 corner: 0.428@0.214)

UserModelView1s30: aPPVC

aPPVC

addSubview: ((IPVCStringEditorView new)

slot: #inputString value: #learnFormula type: #Message;
slot: #modelUpdate value: #learnFormula type: #Message;
slot: #yBM value: #defaultMenu type: #Message)

window: (0 @ 0 extent: 1 @ 1)

viewport: (0.427@0.32 corner: 1.0@0.383)

UserModelView1s31: aPPVC

aPPVC

addSubview: ((IPVCGeneralIOView new)

slot: #label value: 'Default App Increase: ' type: #Literal;
slot: #displayValue value: #defaultApplicationIncrease type: #Message;
slot: #modelUpdate value: #defaultApplicationIncrease type: #Message;
slot: #question value: 'New Default Application Increase' type: #Literal;
slot: #yBM value: nil type: #Literal)

window: (0 @ 0 extent: 1 @ 1)

viewport: (0.351@0.383 corner: 0.683@0.438)

UserModelView1s32: aPPVC

aPPVC

addSubview: ((IPVCGeneralIOView new)

slot: #label value: 'Default Class Increase: ' type: #Literal;
slot: #displayValue value: #defaultClassIncrease type: #Message;
slot: #modelUpdate value: #defaultClassIncrease type: #Message;
slot: #question value: 'New Default Class Increase' type: #Literal;
slot: #yBM value: nil type: #Literal)

window: (0 @ 0 extent: 1 @ 1)

viewport: (0.351@0.438 corner: 0.683@0.486)

UserModelView1s33: aPPVC

aPPVC

addSubview: ((IPVCGeneralIOView new)

slot: #label value: 'Default L2 Trigger: ' type: #Literal;
slot: #displayValue value: #defaultLevel2Trigger type: #Message;
slot: #modelUpdate value: #defaultLevel2Trigger type: #Message;
slot: #question value: 'New Default Level 2 Trigger' type: #Literal;
slot: #yBM value: nil type: #Literal)

window: (0 @ 0 extent: 1 @ 1)

viewport: (0.683@0.383 corner: 1.0@0.438)

UserModelView1s34: aPPVC

aPPVC

addSubview: ((IPVCGeneralIOView new)

slot: #label value: 'Default L3 Trigger: ' type: #Literal;
slot: #displayValue value: #defaultLevel3Trigger type: #Message;
slot: #modelUpdate value: #defaultLevel3Trigger type: #Message;
slot: #question value: 'New Default Level 3 Trigger' type: #Literal;
slot: #yBM value: nil type: #Literal)

window: (0 @ 0 extent: 1 @ 1)

viewport: (0.683@0.438 corner: 1.0@0.486)

UserModelView1s3: aPPVC

aPPVC

addSubview: ((IPVCListView new)

slot: #currentItem value: #currentMethod type: #Message;

slot: #newSelection value: #currentMethod type: #Message;

slot: #itemList value: #currentAvailableMethods type: #Message;

slot: #stringPrint value: false type: #Literal;

slot: #oneItem value: false type: #Literal;

slot: #yBM value: (Menu onString: 'RealMCMenu false top[(MonoStable addMethod Add New Method) (MonoStable

removeCurrentMethod Remove Method)]' type: #Literal)

window: (0 @ 0 extent: 1 @ 1)

viewport: (0.0@0.214 corner: 0.428@0.383)

UserModelView1s4: aPPVC

aPPVC

addSubview: ((IPVCGeneralIOView new)

slot: #label value: 'App Usage: ' type: #Literal;

slot: #displayValue value: #applicationUsage type: #Message;

slot: #modelUpdate value: #applicationUsage type: #Message;

slot: #question value: 'New Application Usage' type: #Literal;

slot: #yBM value: nil type: #Literal)

window: (0 @ 0 extent: 1 @ 1)

viewport: (0.0@0.545 corner: 0.351@0.615)

UserModelView1s5: aPPVC

aPPVC

addSubview: ((IPVCGeneralIOView new)

slot: #label value: 'Class Usage: ' type: #Literal;

slot: #displayValue value: #currentClassUsage type: #Message;

slot: #modelUpdate value: #changeCurrentClassUsage type: #Message;

slot: #question value: 'New Class Usage' type: #Literal;

slot: #yBM value: nil type: #Literal)

window: (0 @ 0 extent: 1 @ 1)

viewport: (0.351@0.545 corner: 0.683@0.615)

UserModelView1s6: aPPVC

aPPVC

addSubview: ((IPVCGeneralIOView new)

slot: #label value: 'Method Usage: ' type: #Literal;

slot: #displayValue value: #currentMethodUsage type: #Message;

slot: #modelUpdate value: #changeCurrentMethodUsage type: #Message;

slot: #question value: 'New Method Usage' type: #Literal;

slot: #yBM value: nil type: #Literal)

window: (0 @ 0 extent: 1 @ 1)

viewport: (0.683@0.545 corner: 1.0@0.615)

UserModelView1s7: aPPVC

aPPVC

addSubview: ((IPVCButtonView new)

slot: #label value: 'Use Method' type: #Literal;

slot: #switchPress value: #useCurrentMethod type: #Message;

slot: #yBM value: nil type: #Literal)

window: (0 @ 0 extent: 1 @ 1)

viewport: (0.683@0.886 corner: 1.0@0.941)

UserModelView1s8: aPPVC

aPPVC

addSubview: ((IPVCButtonView new)

slot: #label value: 'Use Class' type: #Literal;

slot: #switchPress value: #useCurrentClass type: #Message;

slot: #yBM value: nil type: #Literal)

window: (0 @ 0 extent: 1 @ 1)

viewport: (0.351@0.886 corner: 0.683@0.941)

UserModelView1s9: aPPVC

aPPVC

addSubview: ((IPVCButtonView new)

slot: #label value: 'Use Application' type: #Literal;

slot: #switchPress value: #useApplication type: #Message;

slot: #yBM value: nil type: #Literal)

window: (0 @ 0 extent: 1 @ 1)

viewport: (0.0@0.886 corner: 0.351@0.941)

Department methodsFor: PPVCs

DeptView1: aPPVC

"Example Interface 11"

aPPVC isNil

ifTrue: [↑self DeptView1instantiation]

ifFalse:

[self DeptView1s1: aPPVC.

self DeptView1s2: aPPVC.

self DeptView1s3: aPPVC.

self DeptView1s4: aPPVC.

self DeptView1s5: aPPVC.

aPPVC addIPVCLinks: nil]

Department methodsFor: Sub PPVCs

DeptView1Instantiation

```
↑((PPVCView new)
  slot: #yBM value: nil type: #Literal)
```

DeptView1s1: aPPVC

```
aPPVC
  addSubView: ((IPVCStringIOView new)
    slot: #label value: 'Dept Name = ' type: #Literal;
    slot: #displayValue value: #departmentName type: #Message;
    slot: #modelUpdate value: #departmentName type: #Message;
    slot: #question value: 'New Dept Name' type: #Literal;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.0 corner: 1.0@0.282 )
```

DeptView1s2: aPPVC

```
aPPVC
  addSubView: ((IPVCGeneralIOView new)
    slot: #label value: 'No Of Staff = ' type: #Literal;
    slot: #displayValue value: #noOfStaff type: #Message;
    slot: #modelUpdate value: #noOfStaff type: #Message;
    slot: #question value: 'No Of Staff' type: #Literal;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.79 corner: 0.51@1.0 )
```

DeptView1s3: aPPVC

```
aPPVC
  addSubView: ((IPVCStringView new)
    slot: #label value: 'Head = ' type: #Literal;
    slot: #displayValue value: #headName type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.282 corner: 1.0@0.521 )
```

DeptView1s4: aPPVC

```
aPPVC
  addSubView: ((IPVCButtonView new)
    slot: #label value: 'Take on Staff' type: #Literal;
    slot: #switchPress value: #takeOnMember type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.51@0.79 corner: 1.0@1.0 )
```

DeptView1s5: aPPVC

```
aPPVC
  addSubView: ((IPVCStringEditorView new)
    slot: #inputString value: #location type: #Message;
    slot: #modelUpdate value: #location type: #Message;
    slot: #yBM value: (Menu onString: 'RealMCMenu false Top[(MonoStable accept Accept) ]') type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.521 corner: 1.0@0.79 )
```

Person methodsFor: PPVCs

Person1: aPPVC

"Example Interface 12"

aPPVC isNil

ifTrue: [↑self Person1instantiation]

ifFalse:

[self Person1s1: aPPVC.

self Person1s2: aPPVC.

self Person1s3: aPPVC.

self Person1s4: aPPVC.

aPPVC addIPVCLinks: nil]

Person2: aPPVC

"Example Interface 13"

aPPVC isNil

ifTrue: [↑self Person2instantiation]

ifFalse:

[self Person2s1: aPPVC.

self Person2s2: aPPVC.

self Person2s3: aPPVC.

self Person2s4: aPPVC.

self Person2s5: aPPVC.

aPPVC addIPVCLinks: nil]

Person methodsFor: Sub PPVCs

Person1instantiation

```
↑((PPVCView new)
  slot: #yBM value: nil type: #Literal)
```

Person1s1: aPPVC

```
aPPVC
  addSubView: ((IPVCGeneralView new)
    slot: #label value: 'Date Of Birth = ' type: #Literal;
    slot: #displayValue value: #dOB type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.54 corner: 1.0@0.818 )
```

Person1s2: aPPVC

```
aPPVC
  addSubView: ((IPVCStringIOView new)
    slot: #label value: 'Name = ' type: #Literal;
    slot: #displayValue value: #fullName type: #Message;
    slot: #modelUpdate value: #fullName type: #Message;
    slot: #question value: 'Enter New Name' type: #Literal;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.0 corner: 1.0@0.338 )
```

Person1s3: aPPVC

```
aPPVC
  addSubView: ((IPVCStringEditorView new)
    slot: #inputString value: #address type: #Message;
    slot: #modelUpdate value: #address type: #Message;
    slot: #yBM value: (Menu onString: 'RealMEMenu false top[(MonoStable accept Accept) (MonoStable align Align) ]) type:
#Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.338 corner: 1.0@0.54 )
```

Person1s4: aPPVC

```
aPPVC
  addSubView: ((IPVCHorizontalSliderView new)
    slot: #sliderValue value: #age type: #Message;
    slot: #sliderUpdate value: #age type: #Message;
    slot: #sliderRange value: (1 to: 36500) type: #Literal;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.818 corner: 1.0@1.0 )
```

Person2instantiation

```
↑((PPVCView new)
  slot: #yBM value: nil type: #Literal)
```

Person2s1: aPPVC

```
aPPVC
  addSubView: ((IPVCGeneralView new)
    slot: #label value: 'Date Of Birth = ' type: #Literal;
    slot: #displayValue value: #dOB type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.183 corner: 1.0@0.325 )
```

Person2s2: aPPVC

```
aPPVC
  addSubView: ((IPVCStringIOView new)
    slot: #label value: 'Name = ' type: #Literal;
    slot: #displayValue value: #fullName type: #Message;
    slot: #modelUpdate value: #fullName type: #Message;
    slot: #question value: 'Enter New Name' type: #Literal;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.0 corner: 0.446@0.183 )
```

Person2s3: aPPVC

```
aPPVC
  addSubView: ((IPVCStringEditorView new)
    slot: #inputString value: #address type: #Message;
    slot: #modelUpdate value: #address type: #Message;
    slot: #yBM value: (Menu onString: 'RealMEMenu false top[(MonoStable accept Accept) (MonoStable align Align) ]) type:
#Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.446@0.0 corner: 1.0@0.183 )
```

Person2s4: aPPVC

aPPVC

addSubview: ((IPVCHorizontalSliderView new)
slot: #sliderValue value: #age type: #Message;
slot: #sliderUpdate value: #age: type: #Message;
slot: #sliderRange value: (1 to: 36500) type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.0@0.326 corner: 1.0@0.45)

Person2s5: aPPVC

aPPVC

addPPVC: 'DeptView1'
partMsg: #department
variablePPVC: #variablePVC
at: (0.0@0.45 corner: 1.0@1.0)

DetailedPerson methodsFor: PPVCs

DetailedPerson1: aPPVC

"Example Interface 14"

aPPVC isNil

ifTrue: [↑self DetailedPerson1instantiation]

ifFalse:

```
[self DetailedPerson1s1: aPPVC.  
self DetailedPerson1s2: aPPVC.  
self DetailedPerson1s3: aPPVC.  
self DetailedPerson1s4: aPPVC.  
self DetailedPerson1s5: aPPVC.  
self DetailedPerson1s6: aPPVC.  
self DetailedPerson1s7: aPPVC.  
self DetailedPerson1s8: aPPVC.  
self DetailedPerson1s9: aPPVC.  
self DetailedPerson1s10: aPPVC.  
self DetailedPerson1s11: aPPVC.  
self DetailedPerson1s12: aPPVC.  
self DetailedPerson1s13: aPPVC.  
aPPVC addIPVCLinks: nil]
```

DetailedPerson2: aPPVC

"Example Interface 15"

aPPVC isNil

ifTrue: [↑self DetailedPerson2instantiation]

ifFalse:

```
[self DetailedPerson2s1: aPPVC.  
self DetailedPerson2s2: aPPVC.  
self DetailedPerson2s3: aPPVC.  
self DetailedPerson2s4: aPPVC.  
self DetailedPerson2s5: aPPVC.  
self DetailedPerson2s6: aPPVC.  
self DetailedPerson2s7: aPPVC.  
self DetailedPerson2s8: aPPVC.  
self DetailedPerson2s9: aPPVC.  
self DetailedPerson2s10: aPPVC.  
self DetailedPerson2s11: aPPVC.  
self DetailedPerson2s12: aPPVC.  
self DetailedPerson2s13: aPPVC.  
self DetailedPerson2s14: aPPVC.  
self DetailedPerson2s15: aPPVC.  
self DetailedPerson2s16: aPPVC.  
self DetailedPerson2s17: aPPVC.  
self DetailedPerson2s18: aPPVC.  
aPPVC addIPVCLinks: #((16 17 15) )]
```

DetailedPerson methodsFor: Sub PPVCs

DetailedPerson1 instantiation

```
↑((PPVCView new)
  slot: #yBM value: nil type: #Literal)
```

DetailedPerson1s10: aPPVC

```
aPPVC
  addSubView: ((IPVCGeneralView new)
    slot: #label value: 'Pay This Week: ' type: #Literal;
    slot: #displayValue value: #paySoFarThisWeek type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.448@0.646 corner: 1.0@0.751 )
```

DetailedPerson1s11: aPPVC

```
aPPVC
  addSubView: ((IPVCHorizontalSliderView new)
    slot: #sliderValue value: #salIncrease type: #Message;
    slot: #sliderUpdate value: #salIncrease type: #Message;
    slot: #sliderRange value: (1 to: 100) type: #Literal;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.748 corner: 1.0@0.865 )
```

DetailedPerson1s12: aPPVC

```
aPPVC
  addSubView: ((IPVCGeneralIOView new)
    slot: #label value: 'Salary Increase: ' type: #Literal;
    slot: #displayValue value: #salIncrease type: #Message;
    slot: #modelUpdate value: #salIncrease type: #Message;
    slot: #question value: 'New Salary Increase ?' type: #Literal;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.865 corner: 0.457@1.0 )
```

DetailedPerson1s13: aPPVC

```
aPPVC
  addSubView: ((IPVCButtonView new)
    slot: #label value: 'Do Salary increase' type: #Literal;
    slot: #switchPress value: #doSalIncrease type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.457@0.865 corner: 1.0@1.0 )
```

DetailedPerson1s1: aPPVC

```
aPPVC
  addSubView: ((IPVCGeneralView new)
    slot: #label value: 'Date Of Birth = ' type: #Literal;
    slot: #displayValue value: #dOB type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.183 corner: 1.0@0.325 )
```

DetailedPerson1s2: aPPVC

```
aPPVC
  addSubView: ((IPVCStringIOView new)
    slot: #label value: 'Name = ' type: #Literal;
    slot: #displayValue value: #fullName type: #Message;
    slot: #modelUpdate value: #fullName type: #Message;
    slot: #question value: 'Enter New Name' type: #Literal;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.0 corner: 0.446@0.183 )
```

DetailedPerson1s3: aPPVC

```
aPPVC
  addSubView: ((IPVCStringEditorView new)
    slot: #inputString value: #address type: #Message;
    slot: #modelUpdate value: #address type: #Message;
    slot: #yBM value: (Menu onString: 'RealMEMenu false top[(MonoStable accept Accept) (MonoStable align Align) ]') type:
#Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.446@0.0 corner: 1.0@0.183 )
```

DetailedPerson1s4: aPPVC

```
aPPVC
  addSubView: ((IPVCHorizontalSliderView new)
    slot: #sliderValue value: #age type: #Message;
```

slot: #sliderUpdate value: #age type: #Message;
slot: #sliderRange value: (1 to: 36500) type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.0@0.326 corner: 1.0@0.45)

DetailedPerson1s5: aPPVC

aPPVC

addSubview: ((IPVCGeneralIOView new)
slot: #label value: 'Salary:' type: #Literal;
slot: #displayValue value: #salary type: #Message;
slot: #modelUpdate value: #salary type: #Message;
slot: #question value: 'New Salary' type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.0@0.449 corner: 0.448@0.545)

DetailedPerson1s6: aPPVC

aPPVC

addSubview: ((IPVCGeneralIOView new)
slot: #label value: 'Hours This Week:' type: #Literal;
slot: #displayValue value: #hoursSoFar type: #Message;
slot: #modelUpdate value: #hoursSoFar type: #Message;
slot: #question value: 'New hours ?' type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.0@0.545 corner: 0.448@0.64)

DetailedPerson1s7: aPPVC

aPPVC

addSubview: ((IPVCGeneralIOView new)
slot: #label value: 'Yearly Pay:' type: #Literal;
slot: #displayValue value: #yearlySoFar type: #Message;
slot: #modelUpdate value: #yearlySoFar type: #Message;
slot: #question value: 'New Yearly So Far ?' type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.00232@0.64 corner: 0.448@0.748)

DetailedPerson1s8: aPPVC

aPPVC

addSubview: ((IPVCButtonView new)
slot: #label value: 'Week End' type: #Literal;
slot: #switchPress value: #endOfWeek type: #Message;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.448@0.449 corner: 1.0@0.545)

DetailedPerson1s9: aPPVC

aPPVC

addSubview: ((IPVCButtonView new)
slot: #label value: 'Work an Hour' type: #Literal;
slot: #switchPress value: #work type: #Message;
slotArgs: #switchPress value: #(1);
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.45@0.545 corner: 1.0@0.646)

DetailedPerson2Instantiation

↑((PPVCView new)

slot: #yBM value: nil type: #Literal)

DetailedPerson2s10: aPPVC

aPPVC

addSubview: ((IPVCHorizontalSliderView new)
slot: #sliderValue value: #salIncrease type: #Message;
slot: #sliderUpdate value: #salIncrease type: #Message;
slot: #sliderRange value: (1 to: 100) type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.455@0.743 corner: 1.0@0.867)

DetailedPerson2s11: aPPVC

aPPVC

addSubview: ((IPVCGeneralIOView new)
slot: #label value: 'Salary Increase:' type: #Literal;
slot: #displayValue value: #salIncrease type: #Message;
slot: #modelUpdate value: #salIncrease type: #Message;
slot: #question value: 'New Salary Increase ?' type: #Literal;
slot: #yBM value: nil type: #Literal)

window: (0 @ 0 extent: 1 @ 1)
viewport: (0.0@0.865 corner: 0.457@1.0)

DetailedPerson2s12: aPPVC

aPPVC

addSubview: ((IPVCButtonView new)
slot: #label value: 'Do Salary increase' type: #Literal;
slot: #switchPress value: #doSalIncrease type: #Message;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.457@0.865 corner: 1.0@1.0)

DetailedPerson2s13: aPPVC

aPPVC

addSubview: ((IPVCGeneralIOView new)
slot: #label value: 'Work To Do =' type: #Literal;
slot: #displayValue value: #workToDo type: #Message;
slot: #modelUpdate value: #workToDo type: #Message;
slot: #question value: 'New Work To Do ?' type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.448@0.547 corner: 1.0@0.643)

DetailedPerson2s14: aPPVC

aPPVC

addSubview: ((IPVCButtonView new)
slot: #label value: 'Do Work' type: #Literal;
slot: #switchPress value: #doWork type: #Message;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.771@0.643 corner: 1.0@0.747)

DetailedPerson2s15: aPPVC

aPPVC

addSubview: ((IPVCSwitchView2 new)
slot: #label value: '1' type: #Literal;
slot: #initialStatus value: #status1 type: #Message;
slot: #switchOff value: #minus1 type: #Message;
slot: #switchOn value: #plus1 type: #Message;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.55@0.643 corner: 0.627@0.75)

DetailedPerson2s16: aPPVC

aPPVC

addSubview: ((IPVCSwitchView2 new)
slot: #label value: '5' type: #Literal;
slot: #initialStatus value: #status5 type: #Message;
slot: #switchOff value: #minus5 type: #Message;
slot: #switchOn value: #plus5 type: #Message;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.627@0.643 corner: 0.699@0.75)

DetailedPerson2s17: aPPVC

aPPVC

addSubview: ((IPVCSwitchView2 new)
slot: #label value: '10' type: #Literal;
slot: #initialStatus value: #status10 type: #Message;
slot: #switchOff value: #minus10 type: #Message;
slot: #switchOn value: #plus10 type: #Message;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.699@0.643 corner: 0.771@0.75)

DetailedPerson2s18: aPPVC

aPPVC

addSubview: ((IPVCSwitchView new)
slot: #label value: 'SWITCH' type: #Literal;
slot: #status value: #switchStatus type: #Message;
slot: #switchOff value: #switchOff type: #Message;
slot: #switchOn value: #switchOn type: #Message;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.448@0.643 corner: 0.55@0.747)

DetailedPerson2s1: aPPVC

aPPVC

addSubview: ((IPVCGeneralView new)
slot: #label value: 'Date Of Birth =' type: #Literal;

```
slot: #displayValue value: #dOB type: #Message;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.0@0.183 corner: 1.0@0.325 )
```

DetailedPerson2s2: aPPVC

aPPVC

```
addSubview: ((IPVCStringIOView new)
slot: #label value: 'Name = ' type: #Literal;
slot: #displayValue value: #fullName type: #Message;
slot: #modelUpdate value: #fullName type: #Message;
slot: #question value: 'Enter New Name' type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.0@0.0 corner: 0.446@0.183 )
```

DetailedPerson2s3: aPPVC

aPPVC

```
addSubview: ((IPVCStringEditorView new)
slot: #inputString value: #address type: #Message;
slot: #modelUpdate value: #address type: #Message;
slot: #yBM value: (Menu onString: 'RealMEMenu false top[(MonoStable accept Accept) (MonoStable align Align) ]') type:
#Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.446@0.0 corner: 1.0@0.183 )
```

DetailedPerson2s4: aPPVC

aPPVC

```
addSubview: ((IPVCHorizontalSliderView new)
slot: #sliderValue value: #age type: #Message;
slot: #sliderUpdate value: #age type: #Message;
slot: #sliderRange value: (1 to: 36500) type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.0@0.326 corner: 1.0@0.45 )
```

DetailedPerson2s5: aPPVC

aPPVC

```
addSubview: ((IPVCGeneralIOView new)
slot: #label value: 'Salary: ' type: #Literal;
slot: #displayValue value: #salary type: #Message;
slot: #modelUpdate value: #salary type: #Message;
slot: #question value: 'New Salary' type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.0@0.449 corner: 0.448@0.545 )
```

DetailedPerson2s6: aPPVC

aPPVC

```
addSubview: ((IPVCGeneralIOView new)
slot: #label value: 'Hours This Week: ' type: #Literal;
slot: #displayValue value: #hoursSoFar type: #Message;
slot: #modelUpdate value: #hoursSoFar type: #Message;
slot: #question value: 'New hours ?' type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.0@0.545 corner: 0.448@0.64 )
```

DetailedPerson2s7: aPPVC

aPPVC

```
addSubview: ((IPVCGeneralIOView new)
slot: #label value: 'Yearly Pay: ' type: #Literal;
slot: #displayValue value: #yearlySoFar type: #Message;
slot: #modelUpdate value: #yearlySoFar type: #Message;
slot: #question value: 'New Yearly So Far ?' type: #Literal;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.00232@0.64 corner: 0.448@0.748 )
```

DetailedPerson2s8: aPPVC

aPPVC

```
addSubview: ((IPVCButtonView new)
slot: #label value: 'Week End' type: #Literal;
slot: #switchPress value: #endOfWeek type: #Message;
slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.448@0.449 corner: 1.0@0.545 )
```

DetailedPerson2s9: aPPVC

aPPVC

```
addSubview: ((IPVCGeneralView new)
  slot: #label value: 'Pay This Week: ' type: #Literal;
  slot: #displayValue value: #paySoFarThisWeek type: #Message;
  slot: #yBM value: nil type: #Literal)
window: (0 @ 0 extent: 1 @ 1)
viewport: (0.0@0.747 corner: 0.452@0.867 )
```

PVCView methodsFor: PPVCs

pV1: aPPVC

"Example Interface 16"

aPPVC isNil

ifTrue: [+self pV1instantiation]

ifFalse:

[self pV1s1: aPPVC.

self pV1s2: aPPVC.

self pV1s3: aPPVC.

self pV1s4: aPPVC.

self pV1s5: aPPVC.

self pV1s6: aPPVC.

self pV1s7: aPPVC.

aPPVC addIPVCLinks: nil]

PVCView methodsFor: Sub PPVCs

pV1instantiation

```
↑((PPVCView new)
  slot: #yBM value: nil type: #Literal)
```

pV1s1: aPPVC

```
aPPVC
  addSubView: ((IPVCGeneralView new)
    slot: #label value: 'IPVC Type = ' type: #Literal;
    slot: #displayValue value: #yourself type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.0 corner: 1.0@0.1 )
```

pV1s2: aPPVC

```
aPPVC
  addSubView: ((IPVCListView new)
    slot: #currentItem value: #currentSlotName type: #Message;
    slot: #newSelection value: #currentSlotName type: #Message;
    slot: #itemList value: #slotList type: #Message;
    slot: #stringPrint value: true type: #Literal;
    slot: #oneItem value: false type: #Literal;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.0@0.1 corner: 0.4@1.0 )
```

pV1s3: aPPVC

```
aPPVC
  addSubView: ((IPVCStringView new)
    slot: #label value: 'Slot Title = ' type: #Literal;
    slot: #displayValue value: #currentSlotTitle type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.4@0.1 corner: 1.0@0.2 )
```

pV1s4: aPPVC

```
aPPVC
  addSubView: ((IPVCGeneralView new)
    slot: #label value: 'Value = ' type: #Literal;
    slot: #displayValue value: #currentSlotValue type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.399@0.309 corner: 1.0@0.799 )
```

pV1s5: aPPVC

```
aPPVC
  addSubView: ((IPVCStringView new)
    slot: #label value: 'Value Class = ' type: #Literal;
    slot: #displayValue value: #currentSlotValueClass type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.4@0.799 corner: 1.0@0.9 )
```

pV1s6: aPPVC

```
aPPVC
  addSubView: ((IPVCButtonView new)
    slot: #label value: 'Change' type: #Literal;
    slot: #switchPress value: #changeCurrentSlotValue type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.4@0.9 corner: 1.0@1.0 )
```

pV1s7: aPPVC

```
aPPVC
  addSubView: ((IPVCStringView new)
    slot: #label value: 'Comms = ' type: #Literal;
    slot: #displayValue value: #currentSlotDirection type: #Message;
    slot: #yBM value: nil type: #Literal)
  window: (0 @ 0 extent: 1 @ 1)
  viewport: (0.4@0.199 corner: 1.0@0.308 )
```

PVCView class methodsFor: PPVCs

align: aPPVC

"Example Interface 17"

aPPVC isNil

ifTrue: [↑self aligninstantiation]

ifFalse:

[self aligns1: aPPVC.

self aligns2: aPPVC.

self aligns3: aPPVC.

self aligns4: aPPVC.

self aligns5: aPPVC.

self aligns6: aPPVC.

aPPVC addIPVCLinks: nil]

PVCView class methodsFor: Sub PPVCs

alignInstantiation

```
↑((PPVCView new)
    slot: #yBM value: nil type: #Literal)
```

aligns1: aPPVC

```
aPPVC
    addSubView: ((IPVCSwitchView new)
        slot: #label value: 'Left' type: #Literal;
        slot: #status value: #left type: #Message;
        slot: #switchOff value: #verticalOff type: #Message;
        slot: #switchOn value: #leftOn type: #Message;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.0@0.05 corner: 0.3@0.5 )
```

aligns2: aPPVC

```
aPPVC
    addSubView: ((IPVCSwitchView new)
        slot: #label value: 'Centre' type: #Literal;
        slot: #status value: #centre type: #Message;
        slot: #switchOff value: #verticalOff type: #Message;
        slot: #switchOn value: #centreOn type: #Message;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.33@0.05 corner: 0.6@0.5 )
```

aligns3: aPPVC

```
aPPVC
    addSubView: ((IPVCSwitchView new)
        slot: #label value: 'Right' type: #Literal;
        slot: #status value: #right type: #Message;
        slot: #switchOff value: #verticalOff type: #Message;
        slot: #switchOn value: #rightOn type: #Message;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.66@0.05 corner: 1.0@0.5 )
```

aligns4: aPPVC

```
aPPVC
    addSubView: ((IPVCSwitchView new)
        slot: #label value: 'Top' type: #Literal;
        slot: #status value: #top type: #Message;
        slot: #switchOff value: #horizontalOff type: #Message;
        slot: #switchOn value: #topOn type: #Message;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.0@0.55 corner: 0.3@1.0 )
```

aligns5: aPPVC

```
aPPVC
    addSubView: ((IPVCSwitchView new)
        slot: #label value: 'Bottom' type: #Literal;
        slot: #status value: #bottom type: #Message;
        slot: #switchOff value: #horizontalOff type: #Message;
        slot: #switchOn value: #bottomOn type: #Message;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.66@0.55 corner: 1.0@1.0 )
```

aligns6: aPPVC

```
aPPVC
    addSubView: ((IPVCSwitchView new)
        slot: #label value: 'Middle' type: #Literal;
        slot: #status value: #middle type: #Message;
        slot: #switchOff value: #horizontalOff type: #Message;
        slot: #switchOn value: #middleOn type: #Message;
        slot: #yBM value: nil type: #Literal)
    window: (0 @ 0 extent: 1 @ 1)
    viewport: (0.33@0.55 corner: 0.6@1.0 )
```

Appendix I.

User Interface Management System Tool-Set Documentation.

This appendix presents the documentation for the User Interface Management System (UIMS) Tool-set described in chapter six. It lists and describes the individual tools, and shows how they can be used. A set of screen printouts at the end of the appendix shows the relevant tools being used, and are referred to throughout the appendix as figures 1 .. 31.

I.1. Creating New Part Pluggable View Controllers.

A Part Pluggable View Controller (Part PVC) direct manipulation interface can be selected and executed by sending one of two messages to the object being examined :-

- (a) *object plugView*
- (b) *object plugView: PPVCName*

Method (a) presents a list of available Part PVCs (figure 1), and allows the user to select the appropriate one. The user is also given the choice of selecting a new Part PVC, and the default Part PVC. Method (b) is more specific and requires the Part PVC name. Once the correct Part PVC has been selected, the user is prompted for a window to contain the active direct manipulation interface (figures 2 and 3). The user must place the interface window at the required position within the VDU Screen. The Part PVC and Interaction PVC virtual windows are then automatically mapped onto this interface window.

Once the window is located, the appropriate interface is displayed, and the user is allowed to interact with it. If the new Part PVC option was selected, then the user is further prompted for the type of Part PVC required. A menu list containing Special Part PVCs is presented (figure 4), and the user must choose the required type. The user is then prompted to specify the values of any Part PVC Linkage Slots as described in section I.3. While a particular activated Part PVC is being used, the user is prevented from accessing other Smalltalk windows without first closing the active Part PVC. Once the interface is closed, the Part PVC mechanism returns the object to which the Part PVC interface was attached (taking into account any changes which

have been made), and continues executing the next Smalltalk statement, if any.

I.2. Default Construction Menus.

The default Part PVC blue button, or construction menu is shown in figure 5 and provides the following functions:-

'Accept / Close'

This closes down the entire active Part PVC direct manipulation interface.

'Add Pluggable View Controller'

A sub-menu which allows further Interaction PVCs and Part PVCs to be added to this Part PVC.

'Align'

Aligns Interaction PVCs and Part PVCs contained within this Part PVC.

'Change this PPVC'

Allows this Part PVC to be changed to another Part PVC.

'Close this PVC'

Closes this Part PVC, and removes its window from the VDU.

'Generate Code'

Generates the appropriate Part PVC definition for this Part PVC.

'Inspect'

Opens an Inspector window on this Part PVC's Linkage Slots.

'Make this the Default PPVC'

Makes this Part PVC the default Part PVC for the attached object.

'Modify Size'

Modifies position and size of this Part PVC's interface window.

'Re Draw'

Redraws the entire direct manipulation interface. This is useful whenever PVCs have been re-sized and moved around.

'Spawn'

Opens a new Part PVC direct manipulation interface with this Part PVC at the top.

The default Interaction PVC construction menu is shown in figure 6 and provides the following functions:-

'Accept / Close'

This closes down the entire active Part PVC direct manipulation interface.

'Close this IPVC'

Closes this Part PVC, and removes its window from the VDU.

'Generate Code'

Generates the appropriate Part PVC definition for this Interaction PVC's Part PVC.

'Inspect'

Opens an Inspector window on this Interaction PVC's Linkage Slots.

'Links'

A sub-menu which allows Interaction PVC links to be established, broken, and made.

'Add Link'

Enables Interaction PVC links to be established.

'Remove Link'

Interactively removes any of this Interaction PVC's Links.

'Show Links'

Shows any links to this Interaction PVC.

'Modify Size'

Modifies position and size of this Interaction PVC's interface window.

'Re Draw'

Redraws the entire direct manipulation interface. This is useful whenever PVCs have been re-sized and moved around.

Finally, the default Multiple Linkage Interaction PVC construction menu is shown in figure 7. This provides the following extra Interaction PVC functions:-

'Multiple Selectors'

A sub-menu which enable the value contained in a Multiple Linkage Slot to be modified.

'Add Value'

Allows the addition of a new value at the end of the existing list of Multiple Linkage Slot values.

'Change Existing Value'

Allows the modification of the value of an existing Multiple Linkage Slot.

'Remove Existing Value'

Allows the removal of an existing Multiple Linkage Slot value.

'Swap Two Existing Values'

Allows two existing Multiple Linkage Slot values to be swapped around.

Any PVCs which implement their own construction menus must include the appropriate functions from these lists. These functions are now described.

I.3. Adding Interaction Pluggable View Controllers.

An Interaction PVC can be added to a Part PVC at any time. The appropriate Interaction PVC is first selected from the Part PVC construction menu. The

user is then prompted to set the appropriate value for each of the Linkage Slots defined for the new Interaction PVC. The prompting mechanism depends upon whether the Linkage Slot Value Type is set to <Literal>, <Message>, or <Any>. When <Literal> is specified, the *interactiveCreate:on:* Class method for the Linkage Slot Type Value is used. This only allows the user to set the Linkage Slot value to a literal. If <Message> or <Any> is specified one of two menus is displayed from which the user must make a selection. The two menus are shown in figures 8 and 9 respectively. Each menu is divided into two sections. The bottom section lists the available Part PVC object messages which can be used for a particular Linkage Slot. The top section differs depending upon whether <Message> or <Any> is specified. The menu choices provided when <Message> is specified are listed:-

'System Object Messages'

This offers a list of universal messages which are understood by all Smalltalk objects. This list is separated from the object message interface because of its extensive length, and the time it takes to generate the appropriate menu.

'Enter Message Directly'

This allows a message to be directly entered using a text editor window. No checking is done to see whether the message entered is correct.

'Part Message'

This option appears when the object attached to the Part PVC uses the part hierarchy mechanism described in chapter seven. It allows messages to be chosen from the message interface for any of the parts used by the attached object. When this option is selected, the interface designer is prompted to select the appropriate part, and then asked to select the correct message using the same type of construction menu described here.

'None'

This returns a nil value for the Linkage Slot Value.

When <Any> is specified this same set of choices is provided with the addition of one extra choice. This choice, labelled 'Literal Value' allows a

literal Linkage Slot value to be set. This uses the same *interactiveCreate:on:* Class method described above.

After the Linkage Slot values have been correctly specified, the user is finally prompted to place the new Interaction PVC View window within the Part PVC View window to which it is being added. The interface designer may terminate an Interaction PVC addition at any time by pressing the red mouse button outside of the menu.

I.4. Adding Further Part Pluggable View Controllers.

Further Part PVCs can be added to an existing Part PVC by first identifying the part which is to be viewed. The Tool-set automatically lists the available Part PVC object parts, and allows the interface designer to select the appropriate one (figure 10). The available Part PVC descriptions are then displayed in a list for the interface designer to select (figure 1).

The user is then prompted to specify any Linkage Slot values, using the menus described in section I.4. Finally, the user is prompted to place the new Part PVC View window within the Part PVC View window to which it is being added. The interface designer may again terminate a Part PVC addition at any time by pressing the red mouse button outside of the menu.

I.5. Aligning Pluggable View Controllers.

Individual PVCs within the same Part PVC may be aligned with one another. A prompt describing how to use the align feature is first displayed, and the user is asked to confirm their intentions (figure 11). The user must then select the PVC View windows to align. This is achieved by pointing at the appropriate PVC View window, and pressing the red mouse button. The window will then be highlighted. To de-select a PVC window, the process is repeated and the window highlight is removed. When the appropriate PVC windows have been selected, a key on the keyboard must be pressed (figure 12). The user is then presented with a window which contains the aligning information (figure 13). This window is itself generated from an executed Part PVC description, and allows the user to express the type of alignment they require. Once the correct alignments are selected, the user chooses 'Accept / Close' option from the alignment window construction menu. The current interface is then redrawn with the appropriate alignments made.

I.6. Linking Interaction Pluggable View Controllers to one another.

An Interaction PVC can be linked to another Interaction PVC by choosing the 'Add Link' option from the relevant Interaction PVC construction menu. Once chosen, suitable instructions are displayed and the user is asked to confirm their intentions (figure 14). The Interaction PVCs which can be linked to then flash their View windows on the screen. The user must select which Interaction PVCs to link to by pressing the red mouse button while over the required Interaction PVC View window. The link is then made. Further interactions may be required depending upon the effects of linking together specific Interaction PVCs. For example Switch state clashes may occur when adding another switch to an existing switch bank. If an Interaction PVC cannot be linked to any other Interaction PVCs then selecting the 'Add Link' option has no effect. Selecting a non flashing Interaction PVC causes the 'Add Link' function to terminate.

An existing Interaction PVC link can also be broken by choosing the 'Remove Link' option. Again, if no link exists this option has no effect.

Finally, existing Interaction PVC links can be shown by choosing the 'Show Links' option. After displaying a confirmation message (figure 15), any Interaction PVCs linked to the selected Interaction PVC flash on the screen. This flashing will continue until any mouse button is pressed.

I.7. Modifying Pluggable View Controller Size.

The size and position of a PVC View window can be altered at any time. Once the option is selected, the user is prompted to mark the new window area for the relevant PVC (figures 2 and 3). The interface is then redrawn accordingly. Any attempt to place the Interaction PVC or Part PVC View window outside of the bounds of the owner Part PVC is ignored, and the new window size is suitably adjusted to fit into the current owner Part PVC View window.

I.8. Closing Pluggable View Controllers.

An individual PVC may be closed at any time. Doing so removes the relevant PVC from the VDU screen. Each PVC also provides an 'Accept / Close' option which terminates the entire direct manipulation interface. If modification have been made, the user is prompted to confirm their

intentions (figure 16). Given confirmation, the complete active interface is closed, the screen restored, and the attached object which was being viewed / modified is returned.

I.9. Modifying and Reviewing Pluggable View Controller Linkage Slots using the Inspector Window.

The Linkage Slots for a PVC can be altered at run time by choosing the 'inspect' option. This opens an Inspection Window, again generated from a Part PVC description. This inspection window (figure 17) lists the available Linkage Slot names and allows one to be selected. Once selected, the user may view the current Linkage Slot value and change its value by selecting the 'change' button. The Class, and communication direction of the Linkage Slot value is also displayed. Any changes to the Linkage Slot values of a PVC will take affect once the Inspection Window is closed, using the 'Accept / Close' option from one of the Inspector PVC's construction menus.

I.10. Code Generation.

Whenever a new interface is built, or an existing one modified, the user should choose the 'generate code' option to permanently store it. After checking whether the Part PVC has been modified, the user is prompted to enter the new Part PVC name (figure 18). If an existing Part PVC has been modified, then the old name is given and the user allowed to modify it. Once entered, the user is asked whether the Part PVC is to be hidden (figure 19). Answering yes to this question means that the Part PVC will not appear in the available Part PVC list for the object described in section 4.4.1. However, a hidden Part PVC can still be accessed using the *plugView:* message and the correct Part PVC name. The appropriate Part PVC description is then automatically generated and stored as Smalltalk 80 code for future use. If a blank Part PVC name is given, the code generation process is terminated.

I.11. Setting Default Part Pluggable View Controller.

A Part PVC may be made the default Part PVC for an object, or object part, by using the 'Make this the Default PPVC' option from the Part PVC construction menu. The user is then asked to confirm their intentions (figure 20). The new default takes effect immediately.

I.12. Changing Part Pluggable View Controllers.

An existing Part PVC may be changed at any time using the 'Change this PPVC' option. The user is presented with a list of possible Part PVC names (figure 1), and must select the required one. The new Part PVC is then executed, and the appropriate interface displayed using the window area defined for the old Part PVC View.

I.13. Spawning Part Pluggable View Controllers.

A Part PVC may be spawned using the 'Spawn' option. This creates a brand new direct manipulation interface with its own window. The interface is constructed with the spawned Part PVC as the top most Part PVC. Once the 'Accept / Close' option is selected from the new interface, the user is returned to the previous interface window. Attempting to spawn the top most Part PVC in an existing interface is pointless and is prevented.

I.14. Building Extended Lean Cuisine Hierarchic Menus.

Tools are provided for interactively building Extended Lean Cuisine (ELC) Menus. These tools may be used as a result of an input Linkage Slot which allows Literal values, and specifies a <Menu> Type. They may also be used independently by executing one of the following statements :-

- (a) *object editMenu: aMethodName.*
- (b) *object editMenu.*

In each case *object* refers to the Smalltalk object for which an ELC menu is being defined. Method (a) requires the Smalltalk method name where the resulting ELC definition will be stored. Method (b) assumes the default method name of *methodMenu*.

Once the correct statement is executed the Tool-set looks for an existing ELC definition with the appropriate method name. If it does not exist, then the user is asked whether they wish to use the ELC definition from the object's Super-class (figure 20). Again, if no such method exists in the Super-class, the prompt is repeated until the top Super-class is reached. In this case, or if the user specifies not to use the Super-class definition, it is assumed that a new ELC menu is required, and the user is asked to specify the type for the top menu in the hierarchy.

After specifying the top menu, or after executing the existing ELC definition, a menu is displayed showing the available modification functions (figure 22). Once the required menu has been defined, the 'Accept' option should be selected. The user is then requested to confirm that they wish to save the appropriate ELC definition under the specified method name. If this is confirmed, the appropriate definition is store in the objects Class definition as an instance method with the specified name.

The available message interface is constructed from the instance methods which the object can understand. Only messages which require no parameters are allowed, and the appropriate menu offering the available messages is automatically constructed (figure 8). The following functions are provided, and the reader is also referred to a paper by Apperley [Apperley, M.D:1989] for a detailed description of Lean Cuisine Menus.

'Add Above'

Allows another menu item to be added above an existing one. This new item may be a further menu (Non Terminator), or an actual function (Terminator).

'Add Below'

Allows another menu item to be added below an existing one.

'Change One On Status'

Allows a Non Terminator to set, or reset, its One On switch. This switch dictates whether one of the Non Terminator's items must always be selected.

'Change Title'

Allows the title of a menu item to be modified. This title is displayed when the menu is used.

'Delete'

Allows a menu item to be deleted.

'Move Above'

Moves a menu item above another menu item.

'Move Below'

Moves a menu item below another menu item.

'Make New'

Creates a brand new ELC menu. This cancels the existing ELC menu.

'Show / Set initial State'

This displays the current ELC menu that is being defined. The user may then set the initial state accordingly.

'Swap'

This allows two menu items to be swapped.

The user is prevented from performing invalid functions, such as adding two Mutually Exclusive Non-terminator menus with One On switches set, to a single Mutually Exclusive menu. Similarly the user is prevented from swapping two items where one is contained within the other's menu.

Whenever an option is selected which requires that an existing menu item be identified, the current ELC menu is displayed, and the user prompted to select the appropriate item.

Whenever a new item is added, the user is first prompted as to whether the new item is an actual menu (Non-terminator) or function (Terminator) (figure 23). The user is then prompted with an editor window to enter the item's title (figure 24).

If a new menu item is being added, the user is prompted as to whether the menu is Mutually Exclusive, or Mutually Compatible (figure 25). A Mutually Exclusive menu only allows one of its items to be selected at a time, while a Mutually Compatible menu allows any number. The user is then prompted as to whether the menu is Real or Virtual (figure 26). A Real menu appears as an actual sub-menu, while a Virtual menu appears as part of the menu to which it is attached [Apperley, M.D:1989]. Next, the user is asked whether the existing One On status is true or false (figure 27). Finally the user is asked to locate the position in the current ELC menu where the new item is to be added (figure 28).

If a Non-terminator is being added, the user is prompted as to whether the new item is Bistable or Monostable (figure 29). A Bistable Terminator has a state, and requires two method names to be executed whenever the item is selected on, or off. A Monostable has no state, and simply requires a single method name which is executed when the item is selected. The user is then prompted to select the required method name(s) to be executed when the Non-terminator is selected (figure 8). Finally the user is asked to locate the position in the current ELC menu where the new item is to be added (figure 28).

Selecting the 'Accept option' exits the ELC construction facility. The user is first prompted to confirm whether they wish to save the current ELC menu (figure 30). If this is required, then the menu is saved using the method name initially specified. The appropriate ELC description is automatically generated and saved as a Smalltalk method attached to the appropriate Class. This description can then be used to reproduce the ELC menu as required. If the user does not confirm saving the ELC menu no further action is taken, and the ELC menu is not saved.

The user may again terminate any function by pressing the red mouse button outside of any of the menus offered.

Once a new ELC menu definition has been created, it can be used at any time by executing one of the following statements:

- (a) *object plugMenu: menuName.*
- (b) *object plugMenu.*

Where *object* corresponds to the object whose ELC menu is being used, and *menuName* corresponds to the method name which returns the ELC definition. Method (a) requires that the *menuName* be specified, while method (b) assumes a default of *methodMenu*. Once a correct ELC menu definition has been selected, this is executed and the appropriate menu displayed. The top menu offers two choices (figure 31), the first choice is labelled according to the ELC top menu title. The second choice is labelled 'Accept'. Selecting the first choice displays the appropriate ELC menu, and allows functions to be selected from it. Once a function is completed, the user is returned to the top menu again. To leave an ELC menu the second 'Accept' option must be selected. Upon leaving, the value returned by the

last function selected is returned as the executed instantiation statement's value. Any further Smalltalk code is then executed.

In the case of interaction or construction menus used by the proposed UIMS, only the second menu is shown. Selection of specific options then invokes the appropriate PVC behaviour, as determined by the interface designer.

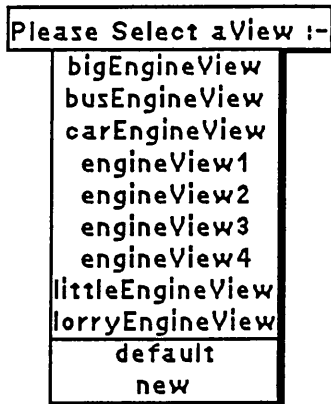


Figure 1



Figure 2

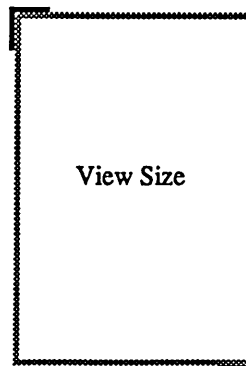


Figure 3

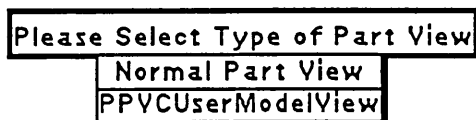


Figure 4

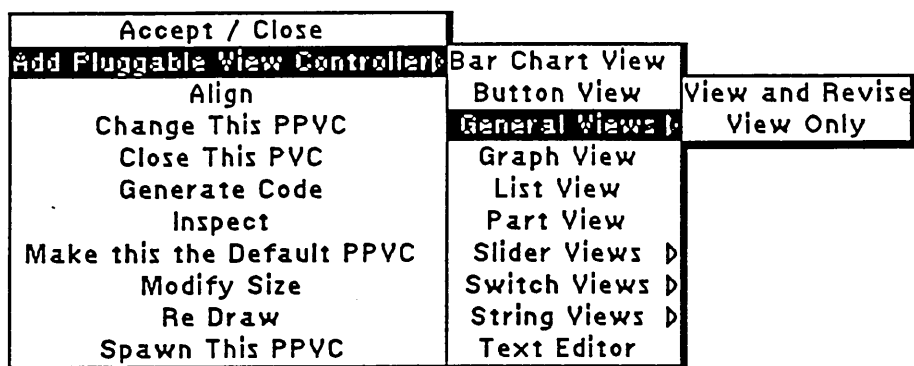


Figure 5

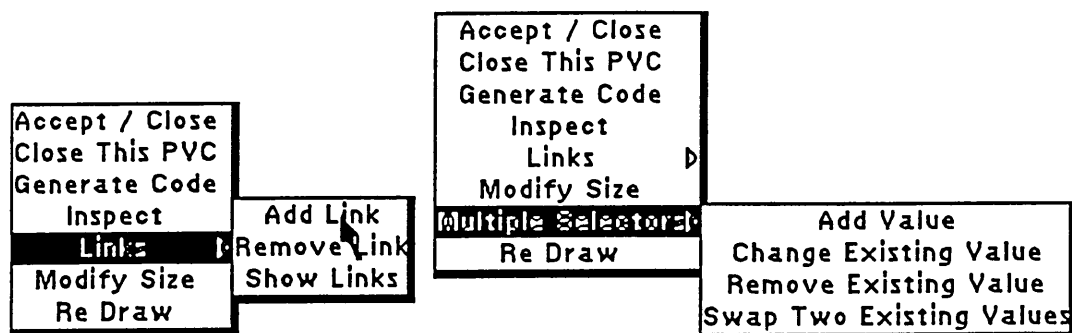


Figure 6

Figure 7

Title Msg	
Literal Value	
System Object Messages	
Enter Message Directly	
Part Message	
None	
availableDescriptions	
defaultView	
description	
engineType	
eType	
initialize	
parts	
power	
size	
sparkPlug	
weight	

Figure 8

Title Msg	
System Object Messages	
Enter Message Directly	
Part Message	
None	
availableDescriptions	
defaultView	
description	
engineType	
eType	
initialize	
parts	
power	
size	
sparkPlug	
weight	

Figure 9

Please select a part	
engine	
wheel1	
wheel2	
wheel3	
wheel4	
None	

Figure 10


ALIGNING - Please select the views to align using the mouse button. Once selected they will be highlighted, to deselect use the mouse button again. Once all of the views are selected press any key. Do you wish to proceed ?	
yes	 no

Figure 11

paul	
Age in Days = 0	dob = 23 October 1989
Salary = 10	Salary + 1 Salary - 1
Pay This Year = 0	Pay This Week = 0
Work an Hour	Hours This Week = 0
End Of Week	

Figure 12

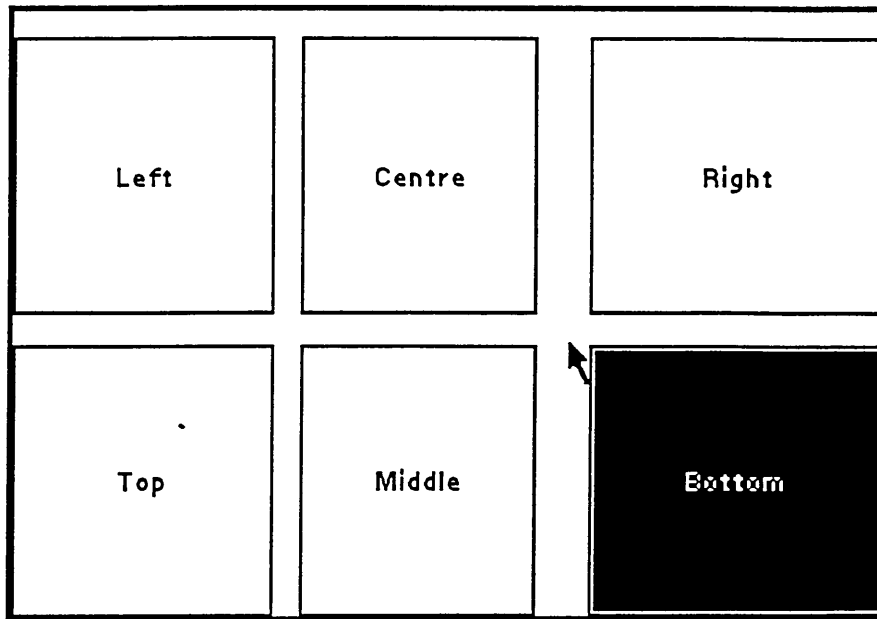


Figure 13


LINKING - The possible views to be linked will flash. Please select the view to link to by using the mouse button. Once selected, the view will be linked. selecting any other view will terminate. Do you wish to proceed ?	
yes	no 

Figure 14


Showing Links - All of my links will flash. To terminate press the mouse button. Do you wish to proceed ?	
yes 	no

Figure 15


Interface has been modified Confirm Close	
yes	no 

Figure 16

IPVC Type = an IPVCListView	
currentItem newSelection itemList stringPrint onItem -----	Slot Title = Interaction Menu
	Comms = Input
	Value = 'Not Specified'
	Value Class = No Value
	Change

Figure 17

Please Type new Title
PersonView1

Figure 18


Is view hidden ?	
yes	 no

Figure 19


Confirm that this view, PersonView1 is default view for the class Person	
yes	 no

Figure 20


No menu exists. Do you require menu from my superclass (yes), or new menu (NO)	
yes	 no

Figure 21

Please Select an option or Quit			
<table border="1"> <tr> <td>Edit Choices:</td> </tr> <tr> <td>Accept</td> </tr> </table>	Edit Choices:	Accept	Add Above Add Below Change One On Status Change Title Delete Move Above Move Below Make New Show / Set initial State Swap
Edit Choices:			
Accept			

Figure 22



Does this item Point to a new Menu		Please Enter title	
yes	 no	<div>NEW ITEM</div> 	

Figure 23

Figure 24


Do You want a Mutually Exclusive (YES) or Mutually Compatible (NO) Menu	
yes	 no

Figure 25


Do You want a Real (YES) or Virtual (NO) Menu	
yes	 no

Figure 26


Must one item always be selected	
yes	 no

Figure 27

Select Menu Item above which NEW ITEM VMC is inserted	
Accept / Close NEW ITEM VMC	
Add PlugViews RME	Bar Chart View
Align	Button View
Change This PPVC	General Views RME
Close This PVC	Graph View
Generate Code	List View
Inspect	Part View
Make this the Default PPVC	Slider Views RME
Modify Size	Switch Views RME
Re Draw	String Views RME
Spawn This PPVC	Text Editor

Figure 28

Do You want a Bistable (YES) or MonoStable (NO)	
yes	no

Figure 29

Do you wish to save this new Menu	
yes	no

Figure 30

Please Select an option or Quit	
TOP	Accept / Close
Accept	Add Pluggable View Controller
	Align
	Change This PPVC
	Close This PVC
	Generate Code
	Inspect
	Make this the Default PPVC
	Modify Size
	Re Draw
	Spawn This PPVC
	Bar Chart View
	Button View
	General Views
	Graph View
	List View
	Part View
	Slider Views
	Switch Views
	String Views
	Text Editor

Figure 31