

Unified Development of Automatically Adapted Interactions

— . —

The Software Engineering Paradigm and a Supporting
Implementation Tool

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT AT CANTERBURY
IN THE SUBJECT OF ELECTRONIC ENGINEERING
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

By
Antonios A. Savidis

February 1998

ABSTRACT

This Thesis introduces a new software interface engineering paradigm, called unified interface development, for constructing interactive software applications that can be automatically adapted to individual user attributes, as well as to the particular contexts of use. The design and implementation processes of the unified development approach are structured according to:

- *The unified interface design method*, which drives the design of dialogues in which diverse user- and usage-context- attribute values (i.e. design parameters) are to be addressed, and
- *The unified interface implementation method*, which provides an architectural framework for “packaging” the implementation of automatically adapted interactions as independent co-operating software components.

The technical profile of the development tools supporting unified development is provided as a list of seven fundamental functional requirements. *The I-GET User Interface Management System* (UIMS) is presented, developed specifically to support unified development, showing how those seven key requirements are practically met, by the supplied implementation mechanisms of the I-GET UIMS.

Finally, existing User Interface development practices are studied and positioned with respect to the unified interface development strategy, and recommendations are given for future work which would promote the unified interface development paradigm.

TABLE OF CONTENTS

LIST OF FIGURES.....	I
ACKNOWLEDGEMENTS	VIII
PUBLICATIONS RESULTING FROM THIS RESEARCH WORK.....	X
FORWARD.....	1
EXECUTIVE SUMMARY.....	3
CHAPTER I : INTRODUCTION.....	6
1.1 TOWARDS USER INTERFACES FOR ALL.....	7
1.1.1 <i>The Emerging Information Society</i>	7
1.1.2 <i>Accessible and High Quality Interaction for All</i>	13
1.1.3 <i>The Mythical Average User</i>	17
1.2 THE CONCEPT OF <i>UNIFIED INTERFACE</i>	19
1.2.1 <i>User- and Usage-Context- Driven Adaptation</i>	20
1.2.2 <i>Dimensions of Adaptation-Targeted Decision Making</i>	21
1.2.3 <i>Adaptability and Adaptivity - the two Dimensions of Interface Adaptation</i>	23
1.2.4 <i>Encapsulation as a Unification Strategy</i>	25
1.3 THE UNIFIED INTERFACE ENGINEERING PARADIGM	28
1.4 RELATED WORK	33
1.4.1 <i>Alternative Access Systems</i>	33
1.4.2 <i>User-Adapted Interaction</i>	40
1.4.3 <i>User Interface Tools</i>	50
CHAPTER II : UNIFIED USER INTERFACE DESIGN METHOD.....	59
2.1 THE NEED FOR POLYMORPHIC ARTEFACTS IN DESIGNING ADAPTED BEHAVIOURS.....	60
2.2 POLYMORPHIC TASK HIERARCHIES	63
2.2.1 <i>Polymorphism versus Mutual Exclusion Operator</i>	65
2.2.2 <i>Polymorphism Entails Decision Making Capability for the Selection of Adaptation Artefacts</i>	66
2.2.3 <i>Categories of Polymorphic Artefacts</i>	69
2.3 POLYMORPHIC TASK DECOMPOSITION PROCESS	71

TABLE OF CONTENTS

2.3.1 From “abstract task design” state.....	72
2.3.2 From “design alternative sub-hierarchies” state.....	73
2.3.3 From “task hierachy decomposition” state.....	73
2.3.4 From “physical task design” state	73
2.3.5 A Process Instance Example for Polymorphic Task Decomposition.....	74
2.4 DESIGNING ALTERNATIVE STYLES.....	75
2.4.1 Identifying Levels of Potential Polymorphism.....	75
2.4.2 Constructing the Space of Decision Parameters	76
2.4.3 Relationships Among Alternative Styles	77
2.4.4 Recording Design Documentation in Polymorphic Decomposition.....	80
2.5 ENGAGING ABSTRACT INTERACTION OBJECTS.....	81
2.5.1 Lexical Role	82
2.5.2 Syntactic Role	83
2.5.3 Semantic Role	83
2.5.4 Re-engineering Designs Through the Role-Based Model.....	84
2.6 INTERFACE DIFFERENTIATION EFFECTS FROM TASK-LEVEL POLYMORPHISM	87
2.6.1 Polymorphism on Top-level Tasks.....	88
2.6.2 Polymorphism on Intermediate Tasks.....	89
2.6.3 Polymorphism on Primitive Tasks	89
2.7 DESIGN SCENARIOS FROM A PROJECT	89
2.7.1 Link Selection Task.....	90
2.7.2 Document Loading Control Task.....	92
2.7.3 Page Browsing Task	92
2.8 DISCUSSION - UNIFIED DESIGN VERSUS DESIGN RATIONALE	94
2.8.1 Design Space Analysis is a Meta-model for Cooperative Design Processes.....	95
2.8.2 Process-oriented versus Artefact-oriented Design Methodologies.....	96
2.8.3 Fusing Unified Design and Design Space Analysis Methods	98
CHAPTER III : UNIFIED USER INTERFACE IMPLEMENTATION METHOD	100
3.1 UNIFIED INTERFACE SOFTWARE ARCHITECTURE	101
3.2 DESCRIPTION OF COMPONENT FUNCTIONAL ROLES	104
3.2.1 User Information Server	104
3.2.2 Context Parameters Server.....	105
3.2.3 Decision Making Component.....	105
3.2.4 Dialogue Patterns Component.....	107
3.3 INTER-COMPONENT COMMUNICATION IN PERFORMING ADAPTATION	107
3.3.1 Adaptability and Adaptivity Cycles.....	108
3.3.2 Detailed Communication Semantics	110
3.4 COMPONENT IMPLEMENTATION ISSUES	116
3.4.1 User Information Server - Implementation Issues	116

TABLE OF CONTENTS

3.4.2 Dialogue Patterns Component - Implementation Issues.....	118
3.4.3 Decision making Component - Impementation Issues.....	120
3.4.4 Context Parameters Server - Implementation Issues.....	121
3.5 DISCUSSION.....	121
3.5.1 Incremental Development.....	121
3.5.2 Reusable Dialogue Patterns.....	122
3.5.3 Employing Componentware Technologies.....	124
3.5.4 Remote Downloading and Application Assembling - the Web Future.....	125
CHAPTER IV : FUNCTIONAL REQUIREMENTS FOR UNIFIED DEVELOPMENT.....	129
4.1 INTRODUCTION.....	130
4.1.1 Key Technical Requirements.....	130
4.1.2 The Seven Key Mechanisms in Unified Development.....	134
4.1.3 The I-GET UIMS Tool.....	135
4.2 METAPHOR DEVELOPMENT.....	141
4.2.1 Metaphor Development Process.....	142
4.2.2 The User-Oriented Nature of Metaphor Development.....	143
4.2.3 The Key Role of Top-Level Containers in Metaphoric Interaction.....	145
4.2.4 A Metaphor Development Case.....	147
4.3 TOOLKIT INTEGRATION.....	150
4.3.1 Previous Related Work.....	151
4.3.2 Implementation Requirements.....	151
4.3.3 Toolkit Integration Support in the I-GET UIMS.....	156
4.4 TOOLKIT AUGMENTATION.....	173
4.4.1 Implementation Requirements.....	174
4.4.2 Designing Augmentation of Windows Object Library with Embedded Scanning Techniques.....	178
4.4.3 Toolkit Augmentation Support in the I-GET UIMS.....	182
4.5 TOOLKIT EXPANSION.....	189
4.5.1 Previous Related Work.....	190
4.5.2 Implementation Requirements.....	192
4.5.3 Toolkit Expansion Support in the I-GET UIMS.....	193
4.6 TOOLKIT ABSTRACTION.....	199
4.6.1 Previous Related Work.....	201
4.6.2 Implementation Requirements.....	203
4.6.3 Toolkit Abstraction Support in the I-GET UIMS.....	207
4.7 STYLE IMPLEMENTATION AND MANIPULATION.....	216
4.7.1 Implementation Requirements.....	216
4.7.2 Style Coordination and Manipulation in the I-GET UIMS.....	218
4.7.3 Mapping the Polymorphic Component Model to Implementation Patterns of the I-GET Language.....	234

TABLE OF CONTENTS

4.8 ARCHITECTURAL OPENNESS	237
4.8.1 <i>The API Gateway for Architectural Extensions</i>	239
4.8.2 <i>The Dialogue Control Gateway</i>	240
4.8.3 <i>Distributed Component-Based Interfaces</i>	242
4.8.4 <i>Toolkit Interoperability</i>	246
4.8.5 <i>Multi-Platform Interactive Software</i>	247
CHAPTER V : SUMMARY, CONCLUSIONS AND FUTURE WORK.....	251
5.1 DISCUSSION.....	252
5.2 FUTURE WORK.....	258
5.2.1 <i>New Interaction Technologies, Metaphors and Toolkits</i>	259
5.2.2 <i>Design Tools for the Unified Design Method</i>	259
5.2.3 <i>Reusable Components of the Unified Software Architecture</i>	260
5.2.4 <i>Extension of the I-GET UIMS for Distributed Dialogue Components</i>	260
5.3 SUMMARY AND CONCLUSIONS.....	262
BIBLIOGRAPHY	268
ANNEX - AN EPILOGUE	287

LIST OF FIGURES

CHAPTER I : INTRODUCTION

Figure 1.1 - Generations in computing (from [Tsichritzis, 1997]).	8
Figure 1.2 - Generations in computer use (from [Stephanidis et al, 1997]).	8
Figure 1.3 - The missing link in an emerging information society (adapted from [Stephanidis et al, 1997a]).	9
Figure 1.4 - Growth of Internet Web sites (from [Nielsen, 1997]).	10
Figure 1.5 - Number of PCs, TV Sets, and Phone Lines per 100 inhabitants in nations of varying income level, 1994 (from [Press, 1997]).	12
Figure 1.6 - Principles of universal design (from [Trace Centre, 1995]).	15
Figure 1.7 - The table on the left is ISO 9241 (Parts 10-17), while the table on the right is ISO 14915. The WD (Working Draft), CD (Committee Draft), DIS (Draft International Standard), and IS (International Standard), correspond to the four stages in the development of standards (ISO / IEC 1992). From [Blanchard, 1997].	16
Figure 1.8 - Engagement (left, traditional development paradigm), as opposed to explicit representation and encapsulation (centre, unified paradigm) of critical design parameters in an iterative development lifecycle (right).	20
Figure 1.9 - Resolving “which” (i.e. driving parameters), “when” (i.e. timing) and “how” (i.e. choice of adaptation artefacts) in the context of the adaptation-oriented run-time decision making process.	22
Figure 1.10 - The complementary roles of adaptability (left) and adaptivity (right) in unified interfaces encapsulating automatically adapted behaviours.	23
Figure 1.11 - Key differences between adaptability and adaptivity in the context of unified interfaces.	24
Figure 1.12 - The general relationships between “design for all”, “encapsulation” and “unification”, when developing solutions towards recurring problem cases.	26
Figure 1.13 - Encapsulation of shared, as well as distinct alternative behaviours, towards abstract artefacts.	28
Figure 1.14 - Key questions raised when addressing the issue of engineering unified interfaces.	29
Figure 1.15 - The dimensions of the development- and resource- domains, in unified interface construction.	31
Figure 1.16 - Logical view of accessibility-oriented adaptations on interactive software, from [Edwards, 1995].	34
Figure 1.17 - The general architectural model of accessibility-oriented environment-level adaptations, adapted from [Savidis, 1994].	35
Figure 1.18 - Goals of adaptation, from [Dieterich et al, 1993].	41

Figure 1.19 - Elements of the structural model for Adaptive User Interfaces, from [Dieterich et al, 1993]. 42

Figure 1.20 - The Seeheim model (left part), and its successor, the Arch model (right part). 52

Figure 1.21 - The MVC model, from [Barkakati, 1991]. 53

Figure 1.22 - The PAC model, supporting hierarchical organisation of agents by connecting to their respective control parts, from [Coutaz, 1990]. 53

CHAPTER II : UNIFIED USER INTERFACE DESIGN METHOD

Figure 2.1 - Basic task operators in the unified design method. 63

Figure 2.2 - The polymorphic task hierarchy concept, where alternative decomposition “styles” are supported (upper part), and an example polymorphic decomposition, for two diverse user groups (blind- and motor-impaired- users, lower part). 64

Figure 2.3 - (A) The use of polymorphism for alternative styles (i.e. dialogue patterns), when those address different user- and usage-context- attributes; and (B) the xor operator for various tasks, when exclusive completion has to be imposed. 66

Figure 2.4 - A design scenario for alternative concurrent sub-dialogues, in order to perform a single task (i.e. task multimodality). 67

Figure 2.5 - Two ways for representing design alternatives when designing for task-level multimodality: (A) via polymorphism, adding run-time control on pattern activation; and (B) via or operator, hard-coding the two alternatives in a singular task implementation. 68

Figure 2.6 - The three artefact categories in the unified method, for which polymorphism may be applied, and how they relate to each other. 69

Figure 2.7 - Representation of alternative physical artefacts: (A) when for the same non-polymorphic task; and (B) when needed due to polymorphic task decomposition. 70

Figure 2.8 - The polymorphic task decomposition process in the unified design method. 72

Figure 2.9 - An example of a polymorphic task decomposition process diagram. 74

Figure 2.10 - An example of a user-profile, as a collection of values from the value domains of user-attributes, from [Savidis et al, 1997g]. 76

Figure 2.11 - Design relationships among alternative styles, and their run-time interpretation. 77

Figure 2.12 - An example of recording design documentation. 80

Figure 2.13 - Role-based model of interaction objects. 82

Figure 2.14 - Producing higher-level design scenarios through the role-based model. 84

Figure 2.15 - The physical design scenario which will be re-engineered. 85

Figure 2.16 - Assigning roles to physical interaction objects. 85

Figure 2.17 - The resulting higher-level object model. 86

Figure 2.18 - An alternative graphical design derived on the basis of the abstract object model. 86

Figure 2.19 - An alternative non-visual Rooms design, derived from the higher-level object model. 87

Figure 2.20 - Levels of task-based polymorphism: (A) for top-level tasks; (B) for intermediate tasks; and (C) for primitive tasks. 88

LIST OF FIGURES

Figure 2.21 - Polymorphic decomposition of the <i>Link Selection</i> task.....	90
Figure 2.22 - Relationships among alternative styles of <i>Link Selection</i> task.....	91
Figure 2.23 - Designed styles for the <i>Page Loading Control</i> task.....	91
Figure 2.24 - Relationships between designed styles for <i>Page Loading Control</i> task.....	92
Figure 2.25 - The Link Summary style, version 1, for <i>Page Browsing</i> task.....	93
Figure 2.26 - The Link Summary style, version 2, for <i>Page Browsing</i> task.....	94
Figure 2.27 - Key differences, concerning the life-cycle of design alternatives, between the Unified Design Method (UDM) and Design Space Analysis (DSA).....	97
Figure 2.28 - The fused process model for combining the Unified Design Method with the Design Space Analysis method.....	98

CHAPTER III : UNIFIED USER INTERFACE IMPLEMENTATION METHOD

Figure 3.1 - The unified interface architecture (vertical dimension), being orthogonal with a typical run-time architecture of interactive systems (Arch meta-model, horizontal dimension).....	103
Figure 3.2 - The User Interface (UI) component manipulation requirements (left), for either <i>adaptability</i> or <i>adaptivity</i> , and their expression via <i>cancellation / activation</i> adaptation actions.....	106
Figure 3.3 - Communication requirements among the components of the unified architecture in order to perform interface adaptation cycles.....	108
Figure 3.4 - Processing steps, engaging communication among architectural components, to perform the initial adaptability cycle (A), as well as the two types of adaptivity cycles (B); the messages communicated among components are labelled with their logical ordering, while internal processing points are indicated with shaded rectangles.....	109
Figure 3.5 - Communicated messages between the User Information Server (UIS) and the Decision Making Component (DMC).....	111
Figure 3.6 - Communicated messages between the User Information Server (UIS) and the Dialogue Patterns Component (DPC).....	112
Figure 3.7 - Communicated messages between the Decision Making Component (DMC), and the Dialogue Patterns Component (DPC).....	114
Figure 3.8 - Communicated messages between the Decision Making Component (DMC), and the Context Parameters Server (CPS).....	115
Figure 3.9 - The architectural pattern for the User Information Server.....	117
Figure 3.10 - Expanding existing interactive software to constitute the Dialogue Patterns Component in unified system architecture.....	118
Figure 3.11 - Semantic gap between design and implementation patterns towards reuse.....	122
Figure 3.12 - The three main content categories of Web documents.....	126
Figure 3.13 - Adaptation-oriented component selection at the Intranet Server, leading to a blending of the unified architecture with intranet-based distributed applications; the mapping to unified architectural components is also indicated.....	127

CHAPTER IV : FUNCTIONAL REQUIREMENTS FOR UNIFIED DEVELOPMENT

Figure 4.1 - The seven key mechanisms for unified development of interactive software, positioned within the three fundamental interaction layers; the order of discussion of each mechanism is also annotated.	135
Figure 4.2 - Run-time architecture of interfaces developed through the I-GET UIMS.....	137
Figure 4.3 - Development roles in the I-GET tool.	139
Figure 4.4 - The four layers of implementation constructs in the I-GET language.....	140
Figure 4.5 - Metaphor development stages in the unified interface engineering paradigm.	142
Figure 4.6 - The four design scenarios demonstrating how top-level containers affect the environment metaphoric properties.	147
Figure 4.7 - Supporting alternative metaphoric representations for container instances in object hierarchies.....	149
Figure 4.8 - An example of a cross -toolkit object hierarchy (WINDOWS, Motif, Athena and Xview objects are mixed).....	155
Figure 4.9 - A dialogue artefact for mixing container objects (coming from different toolkits) which comply to different interaction metaphors.	156
Figure 4.10 - Definition of lexical interaction elements in toolkit interface specification (example from Xaw/Athena toolkit integration).	159
Figure 4.11 - Supported scenarios of use of imported interaction elements for interface development with the I-GET languages.	160
Figure 4.12 - An event handler specification for rubber-banding definition of a line.	161
Figure 4.13 - (1) Mapping, Renaming and Simplification for Xaw/Athena widget set; (2) Simplification for interaction controls from the Windows object library; (3) Transformation and Simplification in the Windows object library; and (4) Generalisation for production of a virtual toolkit (called ViKit) in the integration of Xaw/Athena and Windows object library.....	164
Figure 4.14 - Minimal Object Modelling - definition and explanations.....	167
Figure 4.15 - Structure of messages in the Generic Toolkit Interfacing Protocol: from Dialogue Control to Toolkit Server (left part), and from Toolkit Server to Dialogue Control (right part).	170
Figure 4.16 - Output events for toolkit resource management.	171
Figure 4.17 - Relationship between original- and augmented- toolkit classes in toolkit augmentation.	175
Figure 4.18 - Toolkit augmentation in case that the maximalistic functional requirements are met by a given interface tool.	176
Figure 4.19 - The augmented window management toolbar, presenting three sets of icons (those are display sequentially through the last common icon in each set).	179
Figure 4.20 - The on-screen pop-up virtual keyboard accessible through scanning interaction techniques.	179
Figure 4.21 - Dialogue structure for the augmented scanning dialogue techniques, based on the fundamental <i>SELECT</i> and <i>NEXT</i> actions, for the various Windows objects categories defined.	180

Figure 4.22 - Implementing augmented interaction software at toolkit server side; the Dialogue Control software is not affected, since the transition from the original toolkit library, case (A), to the augmented toolkit library, case (B), takes place at the toolkit server. 183

Figure 4.23 - Injecting augmented dialogue code within toolkit interface specification; dialogue implementations utilising the old toolkit interface specification version need only recompilation. 184

Figure 4.24 - Five examples in which the maximal augmentation features of the I-GET language are utilised for implementing scanning-based dialogues. 186

Figure 4.25 - Interoperation among ActiveX and JavaBeans technologies through Migration Assitant 1.0™ (MA 1.0) and Microsoft Visual J++™ (MVJ++)..... 192

Figure 4.26 - Closure property in maximal expansion - the resulting expanded objects are made through the original dialogue implementation facilities. 193

Figure 4.27 - The two steps in accomplishing toolkit expansion through the I-GET language. 196

Figure 4.28 - Outline of the implementation of a new PushButton class for Xaw/Athena widget set via the I-GET language..... 196

Figure 4.29 - Screen snapshots from expanded object instances, on the Xaw/Athena widget class, implemented via the I-GET UIMS: (1) interactive 2D lines, in which constraints have been applied to force connection among starting- and ending- points; (2) push-buttons, used in a confirm box; (3) enabling / disabling push-buttons (greyed presentation in disabled mode); and (4) row / column containers. 198

Figure 4.30 - An abstract SELECTOR interaction object class, having only two abstract attributes (central diamond); four alternative physical realisations are shown..... 200

Figure 4.31 - Key differences, with respect to polymorphism functional requirements, between OOP development languages and unified interface support. 206

Figure 4.32 - The links between virtual object classes, lexical layers, instantiation relationships, instantiation schemes and lexical object classes, in the I-GET language. 208

Figure 4.33 - Implementation layers for interaction objects in the I-GET language..... 209

Figure 4.34 - (1) Virtual class definitions; (2) instantiation relationship for W95; (3) instantiation relationship for Hawk; (4) A, B and C are resulting virtual-, W95- and Hawk- instances hierarchies, respectively; (5) accessing virtual and lexical attributes; and (6) implementing virtual and lexical methods. 210

Figure 4.35 - The agent class structure: (1) the agent class header with its formal parameters, if the class is to be instantiated synchronously (i.e. via a statement); and (2) agent class header with its instantiation precondition, as well as optional destruction precondition, if the class is to be instantiated asynchronously (i.e. via preconditions). 219

Figure 4.36 - An agent class for implementing a “confirm box” component class, named ConfirmBox, as: (1) precondition-based agent class; and (2) parameterised agent class. Depending on the type of an agent class, instantiation may be caused by making the precondition reach a satisfaction state (see (3), where “confirm” gets the value “true”, in any point in code), or by explicitly calling an agent instantiation (see (4), a “create” statement). 220

LIST OF FIGURES

Figure 4.37 - An implementation scenario in which agent hierarchies are employed.....	222
Figure 4.38 - Run-time picture of agent instances, and their connection with the precondition management strategy of the I-GET UIMS.....	225
Figure 4.39 - The rules for installation or cancellation of preconditions, upon agent instantiation or destruction, respectively.	226
Figure 4.40 - (1) Definition of a shared type and a communication channel; (2) creating a shared object, and sending a message; and (3) a precondition-based agent, with an API-based precondition.	227
Figure 4.41 - (1) Definitions of: (1) a channel to receive adaptation decisions; (2) local structures and functions to store adaptation decisions; and (3) an agent class for receiving and locally storing adaptation decisions.	229
Figure 4.42 - (1) Adding new interactive facilities (child class); (2) adding an adaptive prompting style (toplevel class); (3) the API specifications for receiving monitoring control messages; and (4) adding a monitoring component (child class).	233
Figure 4.43 - Relationships between components which belong in the same set of alternative styles for a polymorphic task (left Table), and relationships among components which correspond to different tasks (right Table).	235
Figure 4.44 - Mapping relationships among components which correspond to different tasks into implementation patterns of the I-GET language.	237
Figure 4.45 - Orthogonal expansion of an I-GET application, towards a unified interface architecture, through the API gateway.	240
Figure 4.46 - Orthogonal expansion of an I-GET application, towards a unified interface architecture, through the Dialogue Control gateway.	241
Figure 4.47 - (1) The <i>distributed toolkit</i> architectural scheme; and (2) the <i>distributed interface</i> architectural scheme.....	244
Figure 4.48 - Blending the distributed component paradigm with the unified interface development architecture.....	245
Figure 4.49 - The merged toolkit server paradigm for importing inter-operating toolkits in the I-GET UIMS.	247
Figure 4.50 - The multi-platform interface development approach in the I-GET UIMS: (a) due to toolkit servers; and (b) due to support for specialisation of physical properties.	249

CHAPTER V : SUMMARY, CONCLUSIONS AND FUTURE WORK

Figure 5.1 - Managing diversity versus implementing diversity in a simple example taken from geometry.	252
Figure 5.2 - Categories of interface development practices.	253
Figure 5.3 - Generating intermediate design layers in semantic-based methods.....	256
Figure 5.4 - Classifying development practices in two groups: those being promising for managing diversity (left column), and those considered as inappropriate for this role (right column).	258

LIST OF FIGURES

Figure 5.5 - Extending I-GET to support distributed dialogue components 262

Figure 5.6 - From diverse users and usage contexts, towards the unified interface engineering
paradigm 263

Figure 5.7 - Three general development process models: (1) emphasis on physical design; (2) equal
emphasis on each level; and (3) emphasis on abstract design. 267

Figure 5.8 - Open questions in designing adapted behaviours 268

ACKNOWLEDGEMENTS

First of all, I would like to acknowledge the continuous valuable support of my supervisors, Professor Michael C. Fairhurst, and Dr. Constantine Stephanidis, during the whole period in which this research work has been carried out.

I would like also to express my appreciation to the Prime Contractor and partners of the TIDE-ACCESS Project, in the context of which the work of this Thesis has been carried out.

Finally, I would like to thank all my colleagues at the Assistive Technology and Human-Computer Interaction Laboratory, for a fruitful collaboration during all these years. Special thanks to my colleague, and also office-room mate for nearly four years, Mr. Demosthenes Akoumianakis, for his willingness in discussing various unconventional technical subjects, many of which did not fall within his own research interests; also, my respect for his analytic and synthetic capabilities, co-operative spirit, and honesty.

Στούς γονείς μου, για την απλότητά τους, την σοφία τους,
και την αγάπη τους

PUBLICATIONS RESULTING FROM THIS RESEARCH WORK

1. Tools for User Interfaces for All. In *proceedings of the 2nd TIDE Congress*, Brussels, Belgium, 26-28 April, 1995, 159-162.
2. Agent Classes for Managing Dialogue Control Specification Complexity. In *proceedings of the 2nd ERCIM Workshop on User Interfaces for All*, November 7-8, Prague.
3. PIM: a Tool for Building Programming Layers on Top of Toolkits. In *proceedings of the 2nd ERCIM Workshop on User Interfaces for All*, November 7-8, Prague.
4. A Generic Direct-Manipulation 3D-Auditory Environment for Hierarchical Navigation in Non-visual Interaction. In *proceedings of the ACM ASSETS'96 conference*, Vancouver, Canada, April 11-12, 1996, 117-123.
5. Unifying Toolkit Programming Layers: a Multi-purpose Toolkit Integration Module. In *proceedings of the Eurographics DSV-IS'97 workshop in Design, Specification and Verification of Interactive Systems*, Granada, Spain (June 4-6). Harrison, M., Torres, J. (Eds). Springer-Verlag, Wien, 1997, 177-192.
6. Augmenting the Windows Object Library with Embedded Scanning Techniques for Motor-Impaired User Access. In *proceedings of the Interfaces 97 Conference, Session M2 - Devices for the Disabled*, May 1997, France, 233-234.
7. Embedding scanning techniques accessible to motor-impaired users in the WINDOWS object library. In *Design of Computing Systems: Cognitive*

- Considerations (21A)*. Salvendy, G., Smith, M., Koubek, R. (Eds). Elsevier, 1997,429-432.
8. Generic containers for metaphor fusion in non-visual interaction: the HAWK interface toolkit. In *proceedings of the Interfaces 97 Conference, Session M2 - Devices for the Disabled*, May 1997, France, 194-196.
9. Unified manipulation of interaction objects: integration, augmentation, expansion and abstraction. In *proceedings of the 3rd ERCIM Workshop on User Interfaces for All*, November 3-4, INRIA Lorraine, 75-89.
10. Designing user-adapted interfaces: the unified design method for transformable interactions. In *proceedings of the ACM DIS'97 conference in Designing Interactive Systems*, Amsterdam, Netherlands, August 18-20, 323-334.
11. Abstract Task Definition and Incremental Polymorphic Physical Instantiation: the Unified Interface Design Method. In *Design of Computing Systems: Cognitive Considerations (21A)*. Salvendy, G., Smith, M., Koubek, R. (Eds). Elsevier, 1997,465-468.
12. Agent classes for managing dialogue control specification complexity: a declarative language framework. In *Design of Computing Systems: Cognitive Considerations (21A)*. Salvendy, G., Smith, M., Koubek, R. (Eds). Elsevier, 1997,461-464.
13. Addressing Cultural Diversity through Unified Interface Development. In *Design of Computing Systems: Cognitive Considerations (21A)*. Salvendy, G., Smith, M., Koubek, R. (Eds). Elsevier, 1997,165-168.
14. Towards the Next Generation of UIST: Developing for All Users. In *Design of Computing Systems: Cognitive Considerations (21A)*. Salvendy, G., Smith, M., Koubek, R. (Eds). Elsevier, 1997,473-476.

15. Unified Interface Development: a Step Towards User Interfaces for All. In *proceedings of the 4rt European Conference for Advancement of Assistive Technology (AAATE'97)*, Porto Carras, 29 September - 2 October 1997, 29-33.
16. Software Architectures for Transformable Interface Implementations: Building User-Adapted Interactions. In *Design of Computing Systems: Cognitive Considerations (21A)*. Salvendy, G., Smith, M., Koubek, R. (Eds). Elsevier, 1997, 453-456.
17. Metaphor Fusion in Non-visual Interaction. In *HCI International'97, Poster Sessions: Abridged Proceedings*, Elsevier, 1997, 61.
18. Encapsulating Multi-Layer Interface Tool Architectures within the Component-Based Development Paradigm. In *HCI International'97, Poster Sessions: Abridged Proceedings*, Elsevier, 1997, 66.
19. Towards User Interfaces for All: Some Critical Issues. Panel session on User Interfaces for All: Everybody, Everywhere, and Anytime. *6th International Conference on Human-Computer Interaction (HCI International'95)*, Tokyo, Japan, July 9-14, Vol 1, 137-142.
20. Towards Multimedia Interfaces for All: a New Generation of Tools Supporting Integration of Design-time and Run-time Adaptivity Methods. In *ACM Multimedia'95 Workshop on Adaptive Technologies for People with Disabilities*, November 11, San Francisco, 1995.
21. Unified Interfaces Development: Tools for Constructing Accessible and Usable User Interfaces. All day tutorial #13, August 26, HCI International'97, San Francisco.

FORWARD

A bus leaves a small city, towards a village in country-side; a young farmer, opens a small portable machine, using a spreadsheet application to check once more that the water supply network he has recently installed in his farm does fulfil his actual farming demands. In the mean time, at another place, in an information kiosk close to an ancient temple, a tourist selects his native language for interaction, then sending an “e-mail postcard” to a friend, including one of the displayed pictures, from the temple he has just visited.

Diversity, can be an ingredient for beauty, giving pleasure and satisfaction to the observers, but may become a real nightmare for the creators of interactive software. The reason is that various alternative ways exist for “telling the same story”, all of which need to be structured in a meaningful way, to be aesthetically pleasing, to convey the correct message, to embody the appropriate mechanics and to smoothly integrate with the environmental context.

Where there is diversity, there is also similarity. They are the two sides of the same coin, each serving a distinctive role in human information processing. To recognise diversity among various artefacts, i.e. identify differences, you should firstly classify those artefacts as being similar, i.e. identify similarities. For this reason, identification of similarities, requiring the extraction of common behavioural patterns among enumerated artefacts, is considered as being a fundamentally “intellectual” process, while the phase of detecting differences, necessitating the juxtaposition of corresponding physical views of subjects, is characterised as being a “mechanistic” process.

Consequently, diversity is the end-result of a process in which similar artefacts are constructed, exposing clearly different physical views with respect to each other, but, still sharing fundamental common properties. The motive for designing towards

diversity may vary from artistic or aesthetic goals, to even physical or practical gains. In some cases there are concrete parameters driving the process of differentiation, while in other cases, it is simply the effect of inspiration or emotional expression by the designer.

Artefacts may expose diversity at various levels, ranging from simple physical parameters, towards attributes which may largely affect the original design. For instance, shoes or clothes have a large number of varying parameters, like size, colour, type, material, etc, which may lead to a large repertoire of alternative models. However, for various customers, the requirements may only be met by physical intervention, i.e. “adaptation”, over one specific instance, and in some cases by the creation of a new custom-made model, i.e. “re-designing”. These are two typical situations when engineered artefacts are to be used: (i) the design is mainly good, however, variation for some specific parameters is not supported; and (ii) the original design is not appropriate at all. The first scenario reveals the need for tailoring some aspects of artefacts, according to the particular demands of use, while the second scenario exposes the necessity of re-designing and re-implementing. As more potential customers are engaged, with different demands, it is likely that new designs will be dictated, while manual adaptations will be largely requested.

While for physical artefacts, the extent to which “adaptations” may be applied is limited, either by temporal or practical factors, software is much more flexible for automatic customisation and individualisation. Software is not a material substance, but it is some form of energy, which can be easily manipulated, transferred, distributed, accumulated, stored, consumed, and used. In order to support the automatic adaptation of interactive software products, there are three main steps needed: (a) to identify the varying user- and usage-context- parameters on the basis of which adaptation will be carried out; (b) to encompass alternative implemented versions of those dialogue patterns which are affected by differing values of the adaptation parameters; and (c) to engineer software products in a way which allows an automatic adaptation process to be carried out when the software is used.

EXECUTIVE SUMMARY

In the development lifecycle of software products and services, the most demanding phase is probably the User Interface engineering process. More than a decade ago, software quality has been mainly attributed to functional characteristics such as operational reliability, efficiency, and robustness. In the last decade, the significant role of usability has been acknowledged, establishing interaction quality as another dimension of software quality.

Today, software is continuously evolving to support human tasks in various new domains, and to facilitate operation from different situations of use. This progressive computerisation of every day activities, gave birth to the concept of an Information Society: *anyone, anytime, anywhere* and for virtually *anything* may take advantage of computer technology. As a result, everybody becomes a potential end-user. In this context, the “User Interfaces for All” objective has been put forward, to ensure that interface accessibility, as well as high interaction quality, will be globally preserved.

When developing interactive software for large user populations, different design parameters emerge, reflecting the diverse user attribute values, as well as the various situations of use (e.g. desk-top systems, public terminals, mobile terminals, hands-free interaction in a car). Inherently, due to different design parameters, alternative design decisions may need to be dictated for various dialogue properties, such as task structure, input syntax, feedback, layout organisation, graphical design, and information visualisation. Hence, potentially, for each individual user, engaged in a particular usage-context, a distinct interface design instance will be required, to ensure interface accessibility and high-quality of interaction. It is evident that, mapping of all those potential design-versions into corresponding software product-versions, requires huge resources; this, would be an impractical engineering strategy.

In this thesis, the concept of a *unified interface* is defined, for interactive software applications and services which are targeted in supporting accessibility and high-quality of interaction for all users, on the basis of the following properties: (a) it packages, into a single software implementation layer, all the various alternative design artefacts which have resulted from the consideration of different design parameters; (b) it encapsulates information regarding individual users (e.g. profiles), as well as usage-contexts; (c) it encompasses decision making mechanisms capable of selecting the most appropriate dialogue artefacts at run-time, given a particular end-user and usage-context; and (d) it encompasses all the necessary implementation mechanisms to co-ordinate and apply such run-time adaptation-oriented decision making, so as to seamlessly provide an adapted interface to each individual end-user.

In this thesis, we will introduce the *unified interface development paradigm* to accomplish the construction of unified interfaces, realising automatically adapted interaction, and we will present a User Interface Management System (UIMS) supporting the implementation of unified interfaces, called the *I-GET UIMS*. The unified development approach provides: (i) the unified design method, a process which drives the design of alternative dialogue artefacts, reflecting diverse design parameters (i.e. user- and usage-context- attribute values); and (ii) the unified implementation method, which directs the packaging of alternative diverse dialogue patterns into a single software system, encompassing automatic adaptation capability.

This thesis consists of five chapters. Chapter I, “Introduction”, introduces the concept of unified interface and discusses relevant work towards accessible and high quality of interaction. Chapter II, “Unified User Interface Design Method”, discusses the unified design technique for organising alternative dialogue patterns, addressing diverse user-and usage-context- attribute values, into a single design structure. The unified design method draws upon the hierarchical task analysis by introducing polymorphism as a fundamental design dimension. Chapter III, “Unified User Interface Implementation Method”, provides the software interface engineering approach for packaging alternative dialogue patterns in an adapted software system implementation. Chapter IV, “Functional Requirements for Unified Development”, establishes the technical profile of tools which are intended to support unified

development, as a list of seven basic functional requirements. Also, the I-GET UIMS is introduced, while the implementation mechanisms it supports are discussed, showing how they satisfy the seven key unified development requirements. Finally, Chapter V, “Summary, Conclusions and Future Work”, starts with discussion regarding existing prevalent interface development practices, judging their overall appropriateness for unified interface development. Then, an outline of future work to be carried out is drawn, for further supporting unified interface development. Finally, an overall summary is provided, together with some important conclusions related to the work reported in this thesis.

Chapter I

INTRODUCTION

1.1 TOWARDS USER INTERFACES FOR ALL

The “User Interfaces for all” [Stephanidis, 1995] objective is a technological target, which reflects as well as follows the principles of *design for all* and *universal access* in Human-Computer Interaction (HCI) in the context of the emerging Information Society. In the context of this thesis, this primary objective is associated with specific technical goals and technological developments. The User Interfaces for All objective is the driving force of this thesis and it has established a new trajectory and scope of work within the domain of User Interface Software and Technology that can be characterised by human-centred principles. There are many related technical domains within the field of HCI which are expected to also contribute towards this goal, by appropriately revisiting currently available methods, techniques, frameworks and tools. The research work reported in this thesis has been mainly concerned with User Interface design and implementation, placing particular emphasis and focusing on the interface software engineering issues.

1.1.1 The Emerging Information Society

The computer technology has a history of about half a century. Just recently this year, ACM and IEEE associations celebrated their 50th anniversary. Both were established just a couple of years after the first working computers came to life, opening the so called “age of computing machines”. Various celebrating events took place in the first half of this year, such as the independent ACM and IEEE 1997 conferences. These two anniversaries have inspired many people to define numerous relevant slogans, all of which can be summarised in something like “*50 years of computing: what next ?*”.

Within those 50 years, progress in computer technology ran with tremendous speed, always bringing changes for continuously enhancing technology, while in certain cases those changes proved to be revolutionary enough, to effectively cause a paradigm shift. People speak today of computer generations to indicate those

significant advancements in computer technology which caused a paradigm shift on the way computers could be utilised and exploited. There have been mainly two complementary approaches in defining computer generations, depending on the role of people manipulating computer technology: (i) *generations in computing*, related to computing professionals [Tsichritzis, 1997], illustrated in Figure 1.1; and (ii) *generations in usage*, related to computer users [Stephanidis et al, 1997], shown in Figure 1.2.

The Fortran-Cobol <i>Card generation</i>
The OS-360-PL1 <i>Mainframe generation</i>
The UNIX-C <i>Minicomputer generation</i>
The MS-DOS-Windows <i>PC generation</i>
The Mail-Web <i>Internet generation</i>

Figure 1.1 - Generations in computing (from [Tsichritzis, 1997]).

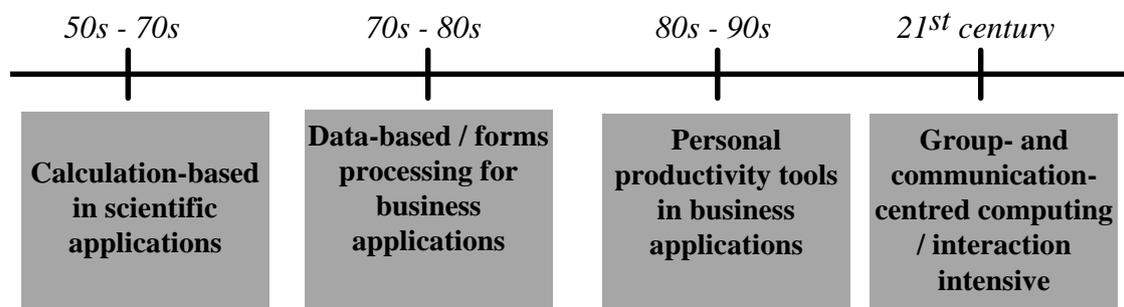


Figure 1.2 - Generations in computer use (from [Stephanidis et al, 1997]).

Irrespective of the different ways in viewing computer generations, there are two important effects resulting from any new generation: (a) the number of people taking advantage of computer technology dramatically increases; and (b) the problems addressed are closer and more relevant to every-day life and working situations. Every new computer generation creates a new status quo, directly affecting the social and professional life of people. This is due to the fact that the appearance of new

added-value technology, aiming to augment the human capability in performing different types of activities and tasks, creates an attitude of potential competition among people.

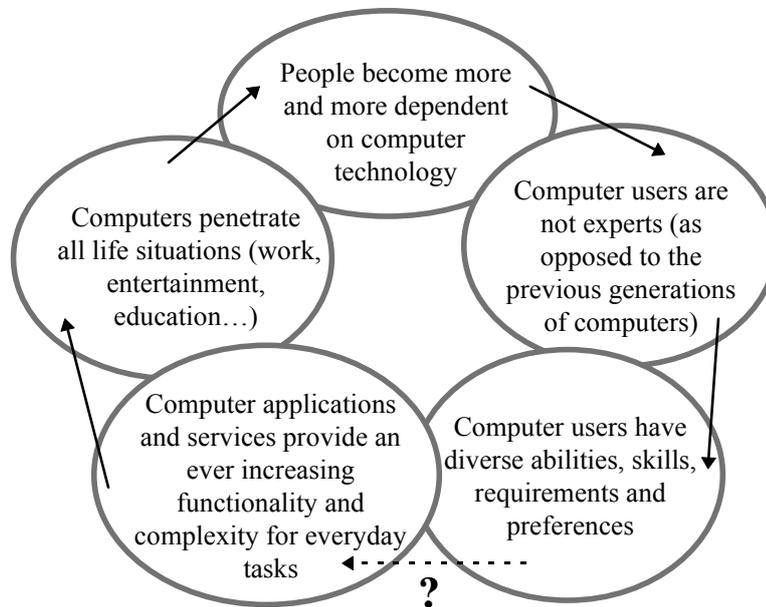


Figure 1.3 - The missing link in an emerging information society (adapted from [Stephanidis et al, 1997a]).

This situation is illustrated in Figure 1.3 [Stephanidis et al, 1997a], which shows how people are not only affected by technology, but may also become dependent on it. Such a dependency may emerge [Perls, 1969] due to external stimulation (e.g. pressure from the working environment, for increasing productivity), or internal stimulation (e.g. a person's instinctual trend for upgrading quality of life). But in the present computer generation, the large potential user populations do not exhibit any particular computer-oriented technological expertise, while they demonstrate diverse abilities, skills, requirements and preferences. This results in a "missing link" in Figure 1.3, indicated via a dashed arrow with a question mark sign on top, which is needed in actually realising what is commonly mentioned as "*bridging technology with people*". This link is necessary in closing the circle, effectively accomplishing the successful establishment and integration of the new computer generation.

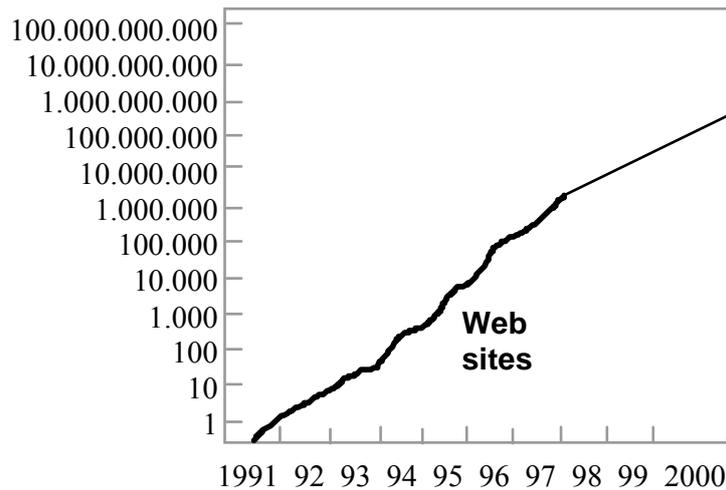


Figure 1.4 - Growth of Internet Web sites (from [Nielsen, 1997]).

Today, six years after the first Web host has been installed world-wide, opening the road to the well known World Wide Web, the number of Web sites has increased with an amazing pace, reaching a growth of a factor of 10 every year [Nielsen, 1997]. This exciting situation is illustrated in Figure 1.4. The explosion of Web sites, providing an open distributed network of information sources, caused a dramatic increase of Web users (also known as “Internet surfers”), thus effectively leading to an increase of computer users. But the vast amount of information sources made available this way, tend to lack an anthropocentric orientation, leading to the need of revisiting the old ways information has been organised, stored and presented:

“In our legal systems, we try to define [i.e. assume] laws of individual behaviour such that not only individuals but society as a whole will behave in the way we hope. The same process will happen as we create an information universe populated with computing engines, but everywhere intimately linked with our “real” world.” [Berners-Lee, 1997]

In the last two years, the Web has considerably matured to support more advanced interaction, capability to manage dynamic information sources, as well as better security, distribution and interoperability. These properties gave birth to new ways of

organising and performing business tasks, as well as establishing communication with customers, leading to *enterprise computing* and *electronic commerce* paradigms respectively. This has introduced new types of sophisticated applications, going beyond the "...dumbwaiter service, shoveling information onto the computer screen faster and faster...[since] Today, the principal use of machines that read Web data is for search engines." [Berners-Lee, 1997]. The distinctive role of HCI in promoting high quality of interaction in such a world-wide network of distributed information and services, has already been acknowledged, especially when considering the remarkable continuous growth of computer users:

"A report on strategic directions in HCI indicates that related challenges include how to serve a much more diverse and less expert user community than that of professional programmers and how to support a wider variety of applications in an environment of increasingly powerful and sophisticated technology" [Adam et al, 1997].

Although there is such a large boost in computer-use due to the revolutionary global software platform offered by the Web, for the largest part of the world population, computers are still seen as machines offering services beyond what is commonly considered as "necessary". In a study [Press, 1997] regarding the availability of major electrical and electronic devices, including personal computers, in nations of varying income (see Figure 1.5), it is clear that computers do not fall in the typical home-equipment shopping list. "It will be many years before digital IT is available on the scale of television, clocks, electric light, and other mind-, value-, and society- altering technologies." [Press, 1997].

This situation has led to more drastic technological approaches: putting together devices such as TV, radio and telephone, by means of integrated services offered by a computer. This idea gave birth to the notion of integrated services through high-speed digital networks, most commonly known as ISDN (Integrated Services Digital Network), and B-ISDN (Broadband ISDN). In Europe, this has led to the

technological initiative for IBC (Integrated Broadband Communications), “as the common vision of the future communications infrastructure” [RACE, 1994], while in the Unites States this concept is known as the “Information Superhighway”.

Income Level	PCs	TV Sets	Phone Lines
Low	0.14	11.8	1.48
Lower Middle	0.72	19.8	8.40
Upper Middle	2.68	24.1	14.14
High	18.26	59.7	51.92
World	4.14	21.7	11.57

Figure 1.5 - Number of PCs, TV Sets, and Phone Lines per 100 inhabitants in nations of varying income level, 1994 (from [Press, 1997]).

Independent manufacturers have already provided their own solutions towards this target; for instance, JavaSoft Inc. has developed the EmbeddedJava™ and PersonalJava™ technologies, allowing Java™ to run in various devices such as “...phones, TV set-tops, Hand-held devices...For the consumer this means...electronics that can access the Web, download content and respond at the touch of a button.” [JavaSoft, 1997]. The most significant result of this initiative until now are the Web phones, which are considered as “intelligent telephones with a touch screen and small keyboard that offer Web browsing...” [JavaSoft, 1997]. Today, we already face the proliferation of software technology in various real-life application domains such as: education, (tele-)communications, information retrieval, work-flow management, banking, shopping, home control, and group collaboration. It is expected that through the emerging technological infrastructure, these application domains will require new capabilities for human-computer interaction, characterised by the following properties, considered as fundamental in the context of an Information Society ([Bangemann, 1994], Esprit LTR Intelligent Information Interfaces (I3) Initiative, <http://www.i3net.org>):

- *Anyone* may take advantage of software applications, hence, the target user population is “broad” with respect to skills, capabilities, requirements and preferences.
- People may access the services of interactive applications from virtually *anywhere*, and *anytime*. This denotes, on the one hand, the need of providing interactive software services from a variety of machinery, ranging from home electronics and public terminals, to high-end PCs and work-stations, while on the other hand the capability of user engagement in various situations of use (e.g. at home, in various organisational settings, while driving).
- People may locate and request services made available for virtually *anything*: work, education, leisure, communication, hobbies, home control and safety, etc.; many new categories of tasks, in which human capacity and capability can be augmented through appropriate Information Technology, are identified and computerised.

1.1.2 Accessible and High Quality Interaction for All

The “User Interfaces for All” objective is the bridge which will enable people to be effectively “immersed” in the emerging Information Society. Given a particular application domain, and a particular software product, potentially anyone should be given the opportunity to use this product. To this effect, accessibility and high quality of interaction are prerequisites. Although accessibility is implied by high-quality of interaction, an intentional “semantic” redundancy is introduced, aiming to emphasise that the first requirement to be satisfied in the software development process is *accessibility for all*, serving as the starting point from which *high-quality for all* in interaction should be also pursued. Next, some general definitions of accessibility and interaction quality are given from [Stephanidis et al, 1997a]; these definitions aim to provide an overall picture of the key issues involved, but they do not constitute a “checklist” of guidelines driving interactive software developments *for all* users:

- The *accessibility* requirement is met for a particular individual user, if for each user-task supported by the interactive software product, there is a corresponding sequence of input actions, which can be performed via user accessible devices, for successfully completing the task; system responses (i.e. feedback) should be provided in output channels engaging accessible devices.
 - The accessibility requirement is similar to the principle of *task completeness* [Markopoulos et al, 1997].
- The *high quality* requirement is fulfilled, if all input action sequences supporting task accomplishment, maximally satisfy the particular individual user abilities, skills, requirements and preferences. Normally, this definition raises the question as to how “maximal satisfaction” can be ensured from the interface design process. The answer is that this definition is not prescriptive, but it indicates that designers should employ any formal (maybe standardised) technique for designing, measuring and asserting high interaction quality, given the particular user parameters, potential contexts of user, tasks and application domain.
 - The high interaction quality requirement is similar to the principle of *task adequacy* [Markopoulos et al, 1997].

The above definitions are not prescriptive, but only identify the boundaries of the problem. In order to start a real development process driven by the “User Interfaces for All” objective, designers should further employ an appropriate methodological prescriptive approach. The two most representative examples of such necessary methods, one for accessibility and one for interaction quality, will be summarised. As it will be concluded from this brief analysis, the user attributes, as *design parameters*, play the key role in making interface design decisions, necessitating the design of alternative artefacts, even for the same user-tasks. Hence, even with the employment of well established (maybe standardised) prescriptive methods, it is unlikely that a single interface design instance (for a product) can be structured, which satisfies the requirements for accessibility and high-quality of interaction for potentially all users.

Principle 1	Equitable use.
Principle 2	Flexibility in use.
Principle 3	Simple and intuitive use, use of the design is easy to understand, regardless of the user's expertise, knowledge, language skills or current concentration level.
Principle 4	Perceptible information.
Principle 5	Tolerance for errors.
Principle 6	Low physical effort.
Principle 7	Size and space for approach and use, appropriate size and space is provided for approach, reach, manipulation, and use regardless of user's body size, posture or mobility.

Figure 1.6 - Principles of universal design (from [TRACE Centre, 1995]).

Regarding accessibility, a lot of work on defining guidelines for universal design has been carried out in the TRACE Research and Development Centre, USA. In Figure 1.6, the seven key principles of universal design are listed (the list is extracted from [TRACE Centre, 1995]; the original list includes detailed explanations which are omitted for clarity). The list ends with a note that "...the principles of universal design in no way comprise all criteria for good design, only universally usable design...other factors are important, such as aesthetics, cost, safety, gender and cultural appropriateness, and these aspects should be taken into consideration as well when designing." [TRACE Centre, 1995]; this note indicates the importance of "traditional" good design (i.e. high quality) principles and techniques, necessary to complement universal design guidelines.

In the context of efforts leading to guidelines for high-quality ergonomic interface design, the ISO Ergonomics Technical Committee has carried out significant work on standardisation, resulting in two concrete outcomes (all presented excerpts from the related ISO standards are taken from [Blanchard, 1997]): (a) ISO 9241 (Ergonomics of Work on Visual Display Terminals - see Figure 1.7); and (b) ISO 14915 (Multimedia User Interface Design - see Figure 1.7).

Part	Subtitle	Status
10	Dialogue principles	IS
11	Guidance on Usability	DIS
12	Presentation of information	CD
13	User Guidance	DIS
14	Menu Dialogues	DIS
15	Command Dialogues	DIS
16	Direct manipulation dialogues	DIS
17	Form-filling dialogues	DIS

Part	Subtitle	Status
1	Introduction and framework	WD
2	General design issues for multimedia controls and navigation	WD
3	Media combination and specific multimedia requirements for individual media	WD
4	Domain specific multimedia aspects	none

Figure 1.7 - The table on the left is ISO 9241 (Parts 10-17), while the table on the right is ISO 14915. The WD (Working Draft), CD (Committee Draft), DIS (Draft International Standard), and IS (International Standard), correspond to the four stages in the development of standards (ISO / IEC 1992). From [Blanchard, 1997].

Currently, ISO 9241 is in a more “stable” form, in comparison to ISO 14915 which has a more recent history. From ISO 9241, the *Guidance on Usability* (Part 11) is an asset of direct relevance to the issue of high interaction quality. An analysis of Part 11, which can be found in [Blanchard, 1997], includes:

“This section emphasises that usability is dependent upon the “context of use”, that the usability of a piece of software is influenced by the specific tasks, users, hardware, physical and social environment, etc.” [Blanchard, 1997].

It is evident, from both categories of guidelines, that users, contexts of use, organisational settings, and the type of input / output equipment available, are critical parameters affecting design decisions. In this sense, it is not possible to define universally applicable design recommendations, unless considering specific instances of these design parameters. The key point here is that the design for diverse user parameter values will necessarily dictate different design decisions; hence, the design

process may potentially end-up with a large number of alternative design instances. Today, software products supply a “single-minded” interface design, based on particular assumptions (by the product developers) regarding the value of each such design parameter. The discussion which will follow in the next Section aims to show that current approaches and common industrial practices consider “average” design parameter values during the design life cycle, and, consequently, lead to poor quality of interaction, or in some cases to inaccessible products.

1.1.3 The Mythical Average User

A commonly used slogan in industrial software development is “designing for an average user”. This philosophy has deep roots into marketing policies, which rely upon the Gaussian distribution model, regarding the value of particular properties of the subject under consideration (i.e. users). One important precondition in the Gaussian model is the presence of an ordering relationship, applicable to the value domain of the properties being considered. For instance, the *height* of human individuals is one such property (i.e. we can order the height of human beings) which can be modelled using the Gaussian model. Following the Gaussian distribution, the average of the global minimum (e.g. shortest man) and global maximum (e.g. tallest man) is a value very close to the corresponding values (e.g. heights) of about 70% of the population.

However, when identification of those human attributes relevant to interaction is needed, the following remarks can be made: (i) there are more than one properties which are important, hence, we should speak of the distribution of multiple varying parameters (e.g. sex, nationality, age, mental abilities, sensory abilities, motor abilities, organisational role, interaction preferences, domain knowledge); (ii) there is not always an ordering relationship between the values of such parameters (e.g. cannot meaningfully order preferences, organisational role, sex, nationality); and (iii) the importance of those parameters may vary dynamically, depending on the value of some specific key parameters (e.g. if “domain knowledge” is, say, “high”, we may

“reduce” the importance that “sex”, “nationality” and “age” may play). When all other important parameters such as context of use, input / output, and social environment are also taken into consideration, we end-up with a large number of key design parameters, say D_1, \dots, D_n (any D_i parameter may either be related to “user”, “context of use”, etc.). Hence, we have $N = |D_1/x/D_2/x \dots x/D_n|$ combinations of such values, each combination defining a particular interaction scenario (i.e. a specific “user”, “context of use”, “social environment”, and “target machine”), likely to require a different treatment in the design process. The number N represents all such distinct design cases, a large number when addressing diverse user attribute values and various contexts of use, while the “average” represents only a single design case, i.e. just one of all those N scenarios.

As we engage more and more relevant parameters, for grouping together items with similar values, the number of resulting groups (of items) is increased, while the size of those groups is decreased (i.e. we are able to “see” more differences). In the case of an infinite number of parameters, we have as many groups as the subjects considered, each group having only a single subject (i.e. each subject is “seen” as different from all others). For instance, assume that we are able to reveal all those parameters defining human personality, and we are driving a design process based on those parameters. It is clear that providing values to such parameters defines only a single individual, while it is completely wrong to speak about an instance of values for those parameters as if representing an “average” personality. In the same manner, the increasing number of design parameters emerging in the context of an Information Society, dictates the need for design processes directly involving all those parameters, as opposed to considering only one (or a few) instance(-s) of values for those parameters (i.e. the “averaging” approach). This last remark is supported by similar observations:

“today’s [approach for] design for the average user results in practical terms to designing for an average [i.e. “low”] interaction quality” [Stephanidis et al, 1997a].

1.2 THE CONCEPT OF *UNIFIED INTERFACE*

In the interface design process, considerations regarding diverse user parameters and different usage contexts point towards the necessity for alternative dialogue patterns. Hence, any run-time adaptation process, if relying upon target user- and usage-context- information, should necessarily utilise and manipulate, from an appropriate implementation resource, the corresponding design knowledge (deciding adaptation) and interaction components (performing adaptation). We define as *unified interface* an interactive system which encapsulates such automatically adapted behaviours, thus, necessarily exhibiting the following properties: (a) it encompasses alternative implemented dialogue patterns, associated to the different values of user- and usage-context- related parameters; (b) it encapsulates representation schemes for user- and usage-context- parameters, while it accesses user- and usage-context- information resources (e.g. repositories, servers); and (c) it embeds design knowledge and decision making capability so as to activate, at run-time, the most appropriate dialogue patterns (i.e. interactive software components), according to particular instances of user- and usage-context- parameters.

A unified interface constitutes a single software system, in the sense that all the various combinations of user- and context- parameters values are addressed by a single (i.e. unified) interactive application. This functional property is very important, since developing multiple interface instances, corresponding to the various enumerated combinations of user- and context- attribute values, is practically unacceptable due to the inherently large development and maintenance costs. The key implementation characteristic of unified interfaces is *encapsulation and representation* of decision parameters (i.e. user- and usage-context- attributes), adaptation design logic and dialogue patterns. Representation in this context implies a formal symbolic notation for communicating design artefacts among designers (e.g. user model), as well as corresponding computable models (e.g. a user profile internal representation) for encapsulating represented artefacts within interactive software applications. The need for explicit representation and encapsulation affects the overall

interface development paradigm, as it is shown in Figure 1.8; in the left-most part, the traditional approach is illustrated, while in the central part, the enhanced development cycle is indicated, as it is required for unified interface development.

Traditional approach	Unified approach	Development lifecycle
Early engagement	Early <i>representation</i>	 Early design
Engagement	Late <i>representation</i>	Late design
Engagement	<i>Encapsulation</i>	 Prototyping
Late engagement	<i>Encapsulation</i>	 Implementation

Figure 1.8 - Engagement (left, traditional development paradigm), as opposed to explicit representation and encapsulation (centre, unified paradigm) of critical design parameters in an iterative development lifecycle (right).

1.2.1 User- and Usage-Context- Driven Adaptation

The “User Interfaces for All” objective reveals the need for designing and implementing for diverse user groups, as well as for varying situations and contexts of use. We emphasise the importance of the usage-context consideration, since, for instance, usage-context parameters may dynamically affect the capabilities of the particular end-user (e.g. while driving, end-users become *functionally* motor-impaired, since hands and feet are occupied for other activities, as well as visually-impaired, since eyes should focus on the road), or provide valuable run-time information on the type of interaction technology available at end-user site (e.g. with low-screen resolution and presence of audio-output hardware, the interface may “decide” to provide less visual effects, while emphasising audio feedback). Hence, our concept of automatically adapted behaviour denotes both user- and usage-context-sensitive adaptation in interactive systems, thus having the following two key properties:

- *User-adapted behaviour*, where the interface is capable of automatically selecting an interaction approach more appropriate to the particular end-user (i.e. *user awareness*);
- *Usage-context adapted behaviour*, where the interface is capable of automatically selecting an interaction approach more appropriate to the particular situation of use (environment context, machine properties - i.e. *usage context awareness*).

The following example scenario clarifies more the idea of automatically adapted interaction in real practice:

Assume that the following user- and usage-context- information is provided to a software interface which encompasses automatically adapted behaviour: (i1) “*low-terminal position*”; (i2) “*high environment noise*”; (i3) “*limited user’s knowledge of the application domain*”; and (i4) “*user is an expert in windowing interaction*”. Then, the interface may decide the following adaptation actions: (a1) arrange primary (i.e. most important) operations horizontally, on top of the screen [addressing i1]; (a2) display secondary functions with large fonts on lower-part of display [addressing i1]; (a3) employ visual feedback, as opposed to auditory [addressing i2]; and (a4) provide adaptive prompting, via pop-up dialogue boxes, for domain-oriented context-sensitive guidance [addressing i3], while employing windowing interaction techniques [allowed due to i4].

1.2.2 Dimensions of Adaptation-Targeted Decision Making

Triggering interface adaptation is a run-time process which requires some kind of an inference capability, in which dialogue design logic, as well as adaptation rationale, are explicitly represented. Such a decision making process should resolve (see also

Figure 1.9): (i) *which* are the driving decision making parameters for adapted interaction (i.e. user- and usage-context); (ii) *when* adaptation may be applied at run-time (i.e. before initiation of interaction, or after interaction has been initiated); and (iii) *how* adaptation is to be realised, by means of specific interaction artefacts, which may concern either the syntactic- or lexical- level of interaction.

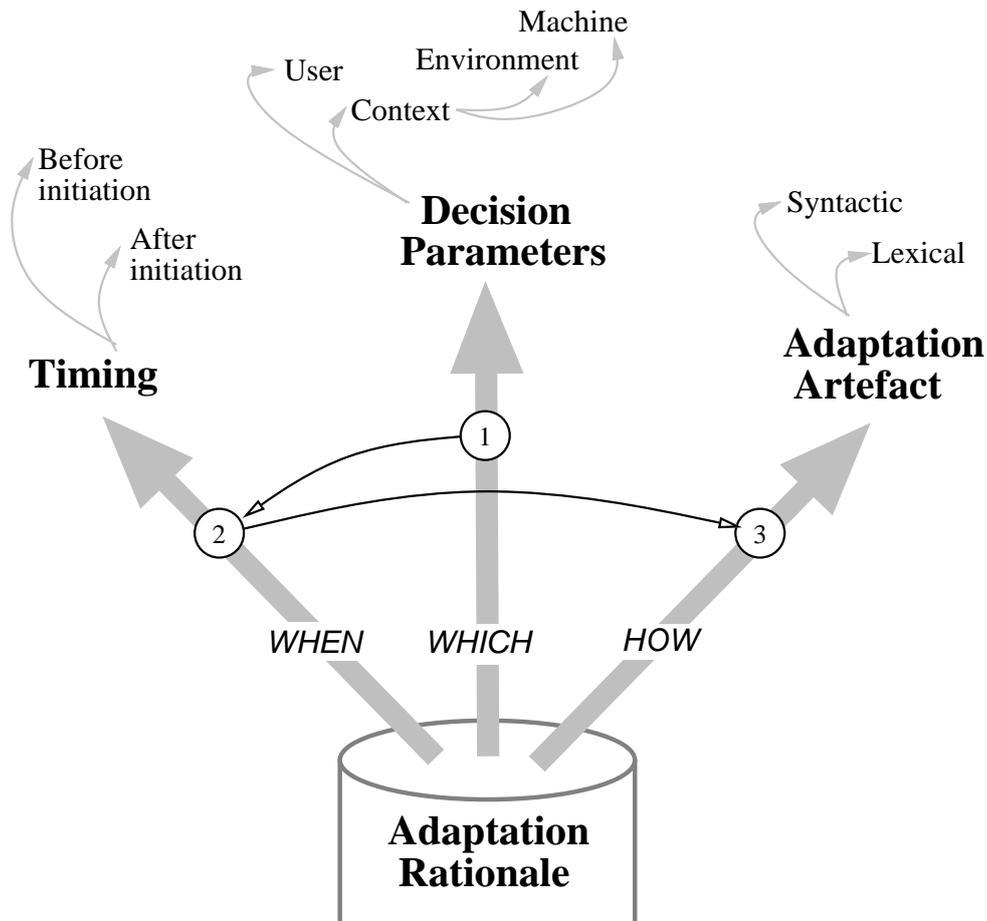


Figure 1.9 - Resolving “which” (i.e. driving parameters), “when” (i.e. timing) and “how” (i.e. choice of adaptation artefacts) in the context of the adaptation-oriented run-time decision making process.

1.2.3 Adaptability and Adaptivity - the two Dimensions of Interface Adaptation

According to the *timing* of performing adaptation, the overall adapted interface behaviour can be seen as the result of two complementary classes of system initiated actions: (a) adaptation actions driven from initial user- and context- information, gained prior to initiating interaction (e.g. physical abilities, domain expertise, terminal capabilities); and (b) adaptation actions decided on the basis of user- and context- information inferred / extracted by monitoring interaction (e.g. inferring that the user gets tired, identifying dynamic user-preferences on a particular interaction style, detecting changes on environment noise).

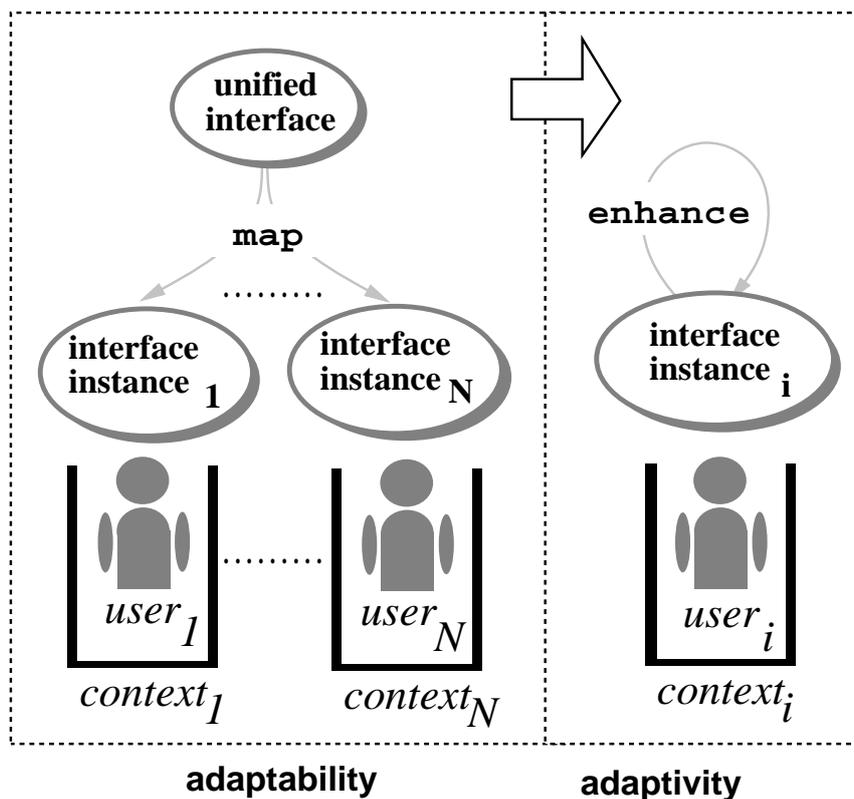


Figure 1.10 - The complementary roles of adaptability (left) and adaptivity (right) in unified interfaces encapsulating automatically adapted behaviours.

The former behaviour is attributed as *adaptability*, reflecting the interface capability to automatically tailor itself initially, to each individual end-user, while the latter behaviour is defined as *adaptivity*, and characterises the interface capability to cope with the dynamically (during interaction) changing / emerging user characteristics and context properties. It should be noted that, if the primary target is accessibility, when developing automatically adapted interactions, the adaptability behaviour is of higher importance, since the essence is to initially (i.e. before initiation of interaction) ensure that a fully accessible interface instance is provided to each particular end-user. In a complementary manner, adaptivity can be only applied on an accessible running interface instance (i.e. user performs interaction), since interaction monitoring is required for the identification of changing / emerging decision parameter values that may drive dynamic interface enhancements. The complementary roles of adaptability and adaptivity approaches are illustrated within Figure 1.10, while the key differences among these two adaptation methods are drawn in Figure 1.11.

Adaptability	Adaptivity
1. User- and usage-context- attributes are considered known prior to interaction.	1. User- / usage-context- attributes are dynamically inferred / detected.
2. “Assembles” an appropriate initial interface instance for a particular end-user and usage-context.	2. Enhances the initial interface instance already “assembled” for a particular end-user and usage-context.
3. Works before interaction is initiated.	3. Works after interaction is initiated.
4. Provides a user-accessible interface.	4. Requires a user-accessible interface

Figure 1.11 - Key differences between adaptability and adaptivity in the context of unified interfaces.

1.2.4 Encapsulation as a Unification Strategy

The idea of encapsulation has emerged in the context of Object Oriented Programming (OOP), as a technique primarily promoting software reuse, while it has been based on the following remarks: (a) there are numerous software parameters which vary in different cases of use (e.g. number / type of elements in a list data structure); (b) writing customised software to address all the various cases of parameter instantiations requires an unacceptable high cost (both for development and maintenance); (c) instead of writing software reflecting hard-coded values of the important parameters, those parameters could be embedded as computable constructs in programs (i.e. be encapsulated), while software could be developed so as to behave accordingly for different parameter values. Making software this way may be harder to start with, but the benefits gained in the long term quickly overcome the required additional initial development effort [Gamma et al, 1995].

Hence, effectively, encapsulation reduces the need for alternative software versions, written for various instances of key parameters, to a single software instance, capable of realising the various alternative functional behaviours. This observation reveals the unification effect of encapsulation, as well as its inherent capability to realise alternative behaviours and allow customised (i.e. adapted) software reuse.

1.2.4.1 Encapsulation as a Means to Attain *Design for All*

In OOP, there are two primary techniques to accomplish the production of highly re-usable software, namely *abstraction* and *encapsulation*. Recently, encapsulation has been acknowledged as playing a major role in supporting software re-use, especially in the context of dialogue patterns [Gamma et al., 1995]; this fact has been also mirrored in the recent developments of the C++ language, where the Standard Template Library (STL) is built mainly around the encapsulation principle, less relying upon class inheritance features. The driving principle of software re-use *is*

providing solutions to recurring software problems, by potentially addressing all the possible cases of use. As a result, developers of re-usable software analyse all potential situations of use, trying to design and encapsulate functional behaviours potentially addressing all possible instances of the particular problem class that the software aims to solve.

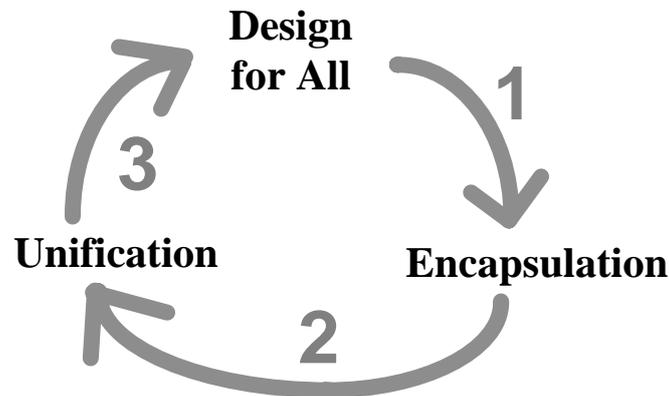


Figure 1.12 - The general relationships between “design for all”, “encapsulation” and “unification”, when developing solutions towards recurring problem cases.

There is a very close association between *design for all*, *encapsulation* and *unification* concepts: (a) conceptually, encapsulation is driven by the design for all principle (see Figure 1.12 - step 1); (b) unification is accomplished by encapsulation of design parameters and alternative behaviours (see Figure 1.12 - step 2); and (c) the design for all objective can be met by unifying the various solution instances for all potential problem cases (see Figure 1.12 - step 3). Those concepts, which have already been beneficially deployed in general purpose software engineering, can offer tremendous benefits in the interface engineering field:

- In general purpose software engineering, accomplishing reusability means developing to address as many problem instances as possible; the specific engineering paradigm applied is encapsulation of varying parameters, resulting in unification of all potential distinct software versions into a single software package.

- Similarly, in interactive software development, applications need to be made potentially accessible by all users, engaged in varying situations of use; hence, on the basis of the user- and usage-context- varying parameters, alternative interactive behaviours need to be constructed, all encapsulated within a unified interface implementation.

The notion of design for all, where “all” attributes to the varying situations of use, has been a key objective in software reuse research, while concepts such as *software adaptability* [Fayad et al, 1996], gained more interest with the appearance of OOP traditions. In the User Interface engineering field, those principles are appropriately exploited in the context of unified interfaces, whereby, given a particular application domain, automatic adaptation for all potential end-users and usage-contexts can be performed.

1.2.4.2 Encapsulation Means *Abstract Design*

The mechanism of encapsulation leads to identification of abstract artefacts. This is a fundamental property, derived from the various steps involved in composing unified artefacts, encapsulating multiple alternative behaviours, from an arbitrary number of distinct common artefacts. In the unification process, which is practically implemented via various encapsulation actions, similarities, as well as differences are captured and appropriately collected and represented. This may result in two categories of behaviours being encapsulated in the unified artefact (see Figure 1.13): (a) *shared behaviours*, which either represent the common aspects of various alternative physical behaviours (their instantiation to physical behaviours is provided by means of encapsulated varying behaviour parameters), or simply constitute physical behaviours which are met across all the various physical artefacts; and (b) *alternative behaviours*, i.e. different real physical behaviours, which provide the alternative physical instantiations of a unified artefact.

The combination of these two behaviour classes leads to the construction of abstract artefacts; on the one hand, these abstract artefacts encompass an abstract behaviour

model (i.e. shared behaviours); on the other hand, they provide parameters to select the specific physical behaviours. This analysis shows, intuitively, how encapsulation is practically a process of abstraction. Hence, abstract design can play an important role in accomplishing the encapsulation of alternative behaviours, the latter being a central technical issue in unified interface development.

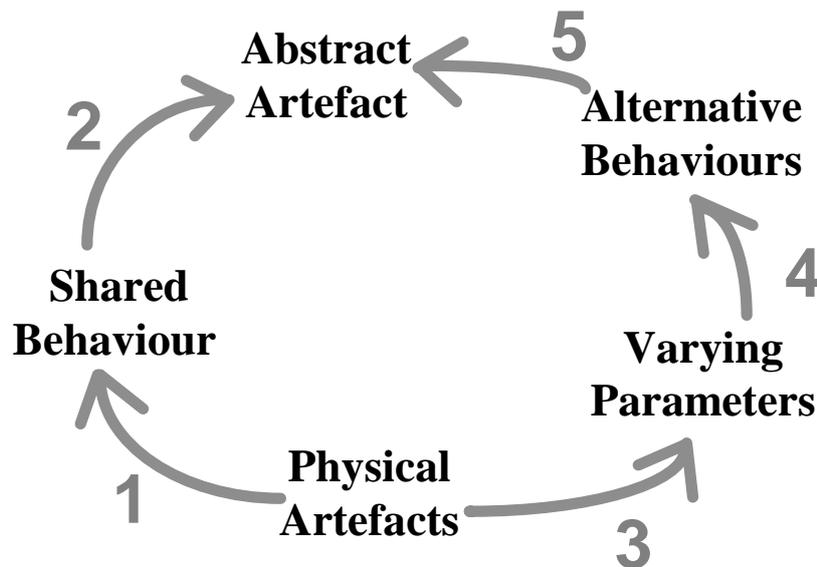


Figure 1.13 - Encapsulation of shared, as well as distinct alternative behaviours, towards abstract artefacts.

1.3 THE UNIFIED INTERFACE ENGINEERING PARADIGM

The unified interface concept introduces several important functional properties, these are: (a) encapsulation of implemented alternative dialogue patterns, reflecting diverse user- and usage-context- properties; (b) encapsulation of user- and usage-context- representation, since those constitute the parameters influencing the decision making in performing adaptation; and (c) decision making capability, for deciding adaptation in a running unified interface, in which user- and usage-context- attributes constitute

input parameters. Consequently, when the issue of engineering unified interface implementation is to be addressed, where engineering involves both designing, as well as implementing the various interactive aspects, some top priority questions are raised (see Figure 1.14).

Q1	How do we <i>design unified interfaces</i> ? Is it possible to directly employ common design practices ?
Q2	In developing software systems, there is a top-priority issue, regarding the envisaged architectural framework [Jacobson et al., 1997]; what is the <i>architectural vision for unified interface implementations</i> ?
Q3	In implementing software systems, it is important to employ “the right tool for the job”; do all interface tools suffice for engineering unified interface implementations, or are there any <i>specific unified development functional requirements</i> ?
Q4	Are there any <i>tools particularly suited in structuring unified interface implementations</i> ? What type of functionality do they provide, for which purpose, in what form (and why in that particular form), and how do they compare to existing interface tools ?
Q5	How do we <i>position existing interface development methods with respect to their overall appropriateness in supporting unified development</i> ?

Figure 1.14 - Key questions raised when addressing the issue of engineering unified interfaces.

These questions reflect pragmatic needs in the overall development process of interactive applications in general, and, they need to be appropriately addressed in the context of unified interfaces. This thesis aims to address all five questions listed within Figure 1.14, relevant to the engineering of unified interfaces, by presenting and discussing the following research outcomes:

- The *unified design method*, being a new design methodology appropriate for driving the design process of unified interfaces; this new design technique introduces the notion of polymorphic design artefacts, which emerge when alternative dialogue patterns have to be designed for various user tasks, according to diverse user- and usage-context- properties.
- The *unified architectural framework*, which provides an implementation paradigm for structuring unified interface implementations. In this context, the detailed system component structure, functional roles, communication interfaces and

implementation issues will be defined. It will be shown that the unified development paradigm possess an *orthogonality* property, with respect to current practices for producing interactive software; thus, it allows the “vertical” growth of existing interactive applications, i.e. without affecting their original software architecture, towards accomplishing automatically adapted behaviours.

- *The unified implementation requirements*, revealing various desirable functional features for interface tools, so as to support unified interface implementation. The unified interface implementation may turn to be a demanding development task, due to the need for implementing and manipulating a potentially large number of alternative interaction artefacts, for diverse user attribute values and usage contexts. Those requirements will help locate the most appropriate development instruments, so as to accomplish specific goals, within unified interface development.
- *The I-GET tool*, which is a language-based User Interface Management System (UIMS), designed and implemented to effectively support the implementation of unified interfaces. It provides the *I-GET 4th Generation Language* (4GL) for interface implementation, which encompasses development features addressing all the various unified development requirements, which are to be discussed in this thesis. The I-GET language provides an effective marriage of general purpose advanced interface construction mechanisms, with special purpose features particularly designed for unified interface implementation support.
- *The assessment of existing development methods for unified development support*. Such an analysis will position prevalent interface development practices with respect to their appropriateness in tackling the various unified interface implementation requirements. As it will be shown, in unified development, multiple tools and diverse development techniques may be employed, since each distinct development tool, or technique, has something different to offer in the overall development process. However, there is still a necessity for a central implementation paradigm, mastering all the various “peripheral” tools and

approaches; through this assessment, the profile of that desirable implementation paradigm will be outlined.

In any type of a production process, there is a development process manipulating elements from a resource domain, leading eventually to a desirable artefact. In this context, the development-, as well as the resource- domains of unified interfaces are illustrated in Figure 1.15.

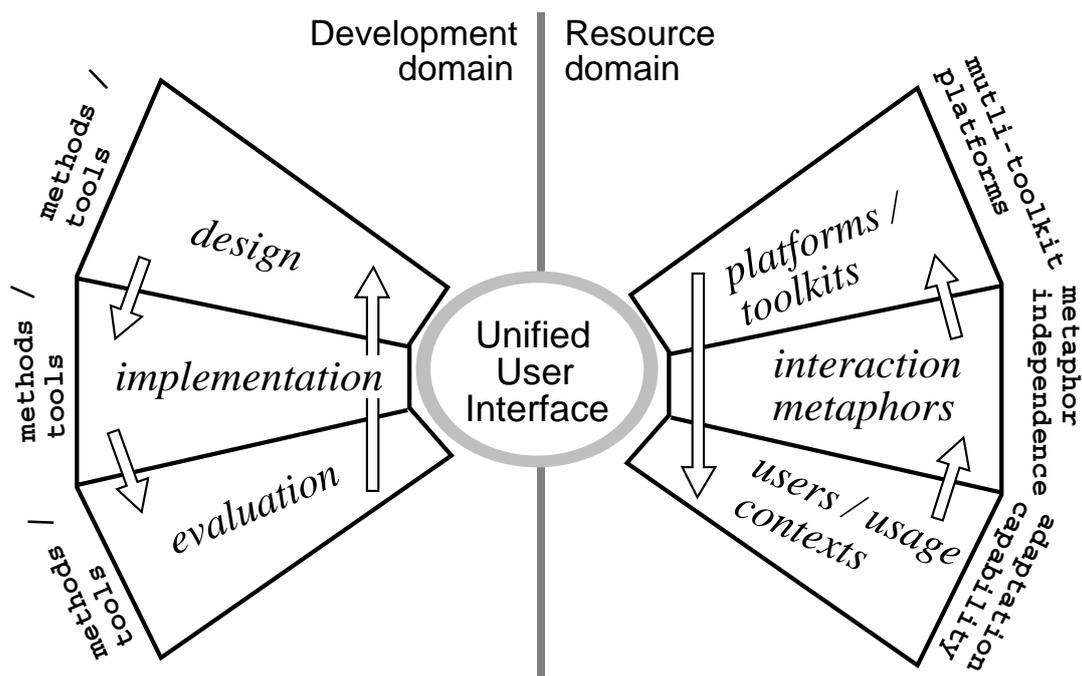


Figure 1.15 - The dimensions of the development- and resource- domains, in unified interface construction.

The development domain encompasses mainly the design, implementation and evaluation phases, for which various types of methods and tools may be supported. In the context of this thesis, the design domain (i.e. a new design technique), as well as the implementation domain (i.e. unified software architecture framework, and unified implementation tool), are to be addressed. The resource domain, emphasises the following properties: (a) it provides the underlying interaction technology (platforms / toolkits [Myers, 1995]); (b) the interaction technology reflects a systematic design of dialogue metaphors (e.g. desk-top windowing metaphor); and (c) all dialogue

artefacts are made to reflect particular user parameters (e.g. desk-top metaphor designed for sighted office worker [Canfield et al, 1982]), as well as contexts of use (e.g. graphics high-resolution terminal in an office establishment).

It should be noted that users, as well as usage contexts, are treated as resource parameters. This fact emphasises the paradigm shift that unified interfaces introduce, moving away from interactive systems reflecting single-minded designs and hard-wired user- and usage-context- attribute values, towards dialogues *encapsulating* those parameters, providing multiple adapted interface instances.

The unified development process requires both the necessary methods and tools from the development domain, as well as the required resources from the resource domain. Even though tools play a central role in determining whether unified development may, or may not be supported, there are situations in which unified development may not be enabled simply because the necessary resources are missing. A typical instance of this situation has been the case of dual interface development [Savidis et al., 1995a], seen as an instantiation of the unification paradigm for two specific user categories (i.e. sighted and blind people); although a tool, called HOMER [Savidis et al., 1998], has been constructed to support dual interface construction, the lack of non-visual interaction elements posed the explicit implementation of a non-visual toolkit (COMONKIT toolkit, implementing the Rooms non-visual metaphor [Savidis et al., 1995b]). Hence, progress in both domains is crucial so as to ensure that, on one hand, the necessary interaction technology is made available (for various users and usage-contexts), while on the other hand, that the available development instruments do support efficiently and effectively the manipulation of interaction technologies, within a unified development process.

1.4 RELATED WORK

Previous work, potentially aiming towards accessible and / or high quality of interaction for all users, falls in three general categories: (a) Accessibility-oriented developments, in which the driving goal has been to facilitate automatic access for disabled users to interactive applications originally developed for able users; such work will be referred to as alternative access systems. (b) Developments targeted in enabling the User Interface to dynamically adapt to end-users, by inferring particular user attribute values during interaction (e.g. preferences, application expertise); additionally, there have been more recent efforts towards the automatic configuration of lexical interaction features on the basis of user models. Work which falls in this category will be referred to as user-adapted interaction. (c) Development efforts related to User Interface tools, concerning both prevalent architectural practices for interactive systems, as well as interface tools supporting development of dialogues accessible directly by potentially “all” users.

1.4.1 Alternative Access Systems

There are two possible technical routes in alleviating the accessibility problem to interactive software products that are not made accessible from the beginning: One is to take each application separately and make all the necessary implementation actions so that an alternative accessible interface is finally produced (i.e. *product level adaptation*). The other is to “intervene” at the level of the particular interactive application environment (e.g. Windows 95, X Windowing System) in which the particular inaccessible application is deployed, and produce appropriate software and hardware technology so as to make that environment alternatively accessible (i.e. *environment level adaptation*); in effect, not only the subject interactive application will now become accessible, but, potentially, all the rest applications running through that interactive environment.

Product-level adaptation has been practically tackled as re-development from scratch; until now, it has proved to be so resource demanding that this approach is considered as the least favourable strategy for alternative access. As a result, environment-level adaptations, addressing a range of applications, has been acknowledged as a more promising strategy:

“Ideally one adaptation should be available for a range of applications. This makes the best sense economically for the adaptation only need be carried out once, and each user need buy just one version of it. The alternative of building a range of adapted applications (a talking word processor, a talking spreadsheet, and a talking database, for instance) represents wasteful duplication.”, from [Edwards, 1995].

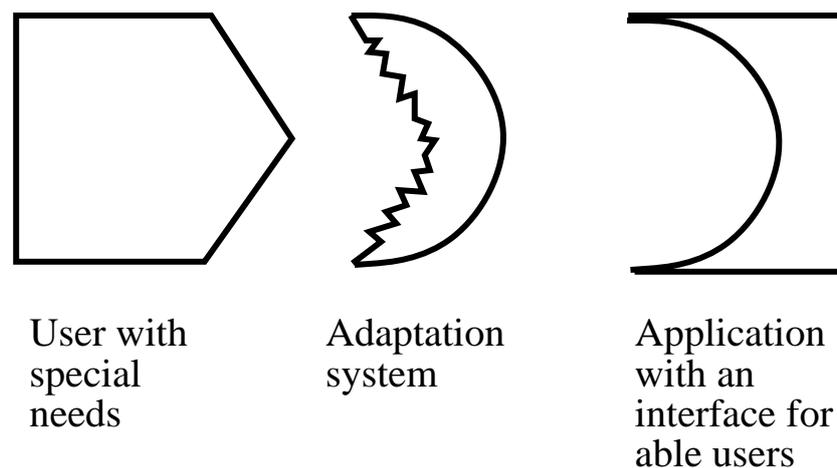


Figure 1.16 - Logical view of accessibility-oriented adaptations on interactive software, from [Edwards, 1995].

The logical view of alternative access methods, either for product-level or environment-level adaptations, is illustrated in Figure 1.16 (notice that there is still not a perfect match between the implemented adaptation and the intended user group), while the architectural view of environment-level adaptations is outlined in Figure 1.17 (*server* being the environment-supported interactive functionality, while *client* denotes running interactive applications).

Approaches falling in the category of environment-level adaptations may rely upon well documented and operationally reliable software infrastructures, supporting effective and efficient extraction of dialogue primitives during user-computer interaction. Such dynamically extracted dialogue primitives are to be reproduced, at run-time, to alternative input / output forms, directly supporting user access. Examples of such software infrastructures are the Active Accessibility™ Technology, by Microsoft, and the Java Accessibility™ Technology, by JavaSoft.

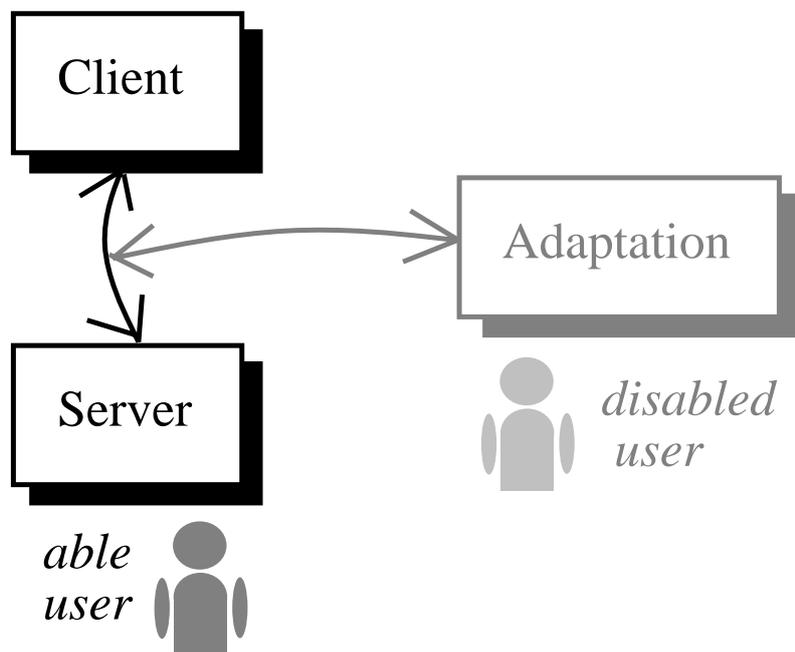


Figure 1.17 - The general architectural model of accessibility-oriented environment-level adaptations, adapted from [Savidis, 1994].

It is argued that automatic interface reproduction is of questionable interface quality, since the explicit interface design, implementation, and evaluation cycles are excluded; this issue has been already identified in the context of non-visual interaction, where, according to [Savidis et al, 1995a], “non-visual User Interfaces should be more than automatically generated adaptations of visual dialogues”, as well as a general theoretical and practical problem of environment-level adaptations:

“...much of the design effort is concerned with adapting existing interfaces. The question is often asked as to why one should adapt an existing interface rather than design one from scratch with the needs of disabled users in mind”, from [Edwards, 1995].

This last observation leads us back to the issue and arguments regarding product-level adaptations. Two alternative answers exist: (a) if we continue to follow an approach in which mainstream software products are developed following considerations for the “average” user, then we would need to *reactively* develop solutions for making all those products available to disabled users; and (b) if there is a proactive approach to the requirements of disabled users, then the software production process will need to be re-considered. The unified development paradigm is an interface engineering approach following a proactive strategy; It will be shown in this thesis that this approach caters for accessibility and quality of interaction and that it is also practically cost effective.

A detailed survey of most the significant environment-level adaptation systems follows, dealing mainly with developments providing access to blind or motor impaired people. We will mainly focus attention on the more recent developments, highlighting their distinctive capabilities.

1.4.1.1 Environment-level adaptations for blind users

The GUIB Project. The GUIB Project (Textual and Graphical User Interfaces for Blind People) [GUIB Project, 1995], partially funded by the TIDE Programme of the European Commission (DG XIII), has focused on the two widely used interactive environments, i.e. MS-WINDOWS and X WINDOWING SYSTEM. One common technical line has been taken for both environments, though being applied through different implementation mechanisms, addressing the following technical issues: (a) filtering of the various visual interaction elements displayed on the screen at run-time; (b) construction of an internal representation of the display structure, called an Off-Screen-Model (OSM); (c) reproduction of the graphical windowing interface structure

to an appropriate textual windowing structure displayed on a separate PC, being accessible via a text-based screen reader and non-visual input / output devices; (d) within the non-visual dialogue, via the text-based interface instance, all blind user actions cause artificial input to the visual interface instance, as if a sighted user would perform those actions on the windowing interface; and (e) windowing interface updates, are propagated back to the textual interface, via the OSM model, thus keeping the textual interface instance consistent with the windowing version.

Some additional technical points, regarding the GUIB Project, are: (i) Graphical structures, like pictures, icons, and visual representations, cannot be filtered, since they are represented via lower-level elements (e.g. pixel arrays or lists of geometric primitives); more sophisticated methods, like image understanding, cannot be relied upon, on the one hand because they are not capable for coping with such demanding problems, while on the other hand, there is the problem of *presentational ambiguity* (i.e. common symbolisms referring to different semantics, [Savidis, 1994]). (ii) The desk-top windowing metaphor is explicitly transferred to the non-visual interaction domain, presumably assuming that such a visually-oriented metaphor could be also appropriate for blind users; however, this hypothesis has attracted considerable criticism[Savidis et al., 1995b].

The Mercator Project. The Mercator Project [Mynatt et al, 1994], [Edwards et al, 1994], has driven developments in making the X WINDOWING SYSTEM client applications accessible by blind people. The key technical points of this work are: (a) filtering interaction elements from applications on the basis of the Editres protocol; (b) construction of an Off-Screen-Model; (c) selective reproduction of the OSM to a 3D-auditory structure called AudioRooms [Mynatt et al, 1995]; and (d) propagation of blind-user actions to the visual interface, and mapping of visual interface updates to the non-visual audio structure. The technical approach taken in the Mercator project, regarding filtering and OSM construction, is similar to that of the GUIB Project, and complies to the general architectural model of Figure 1.16 (in Figure 1.16, the OSM is not explicitly represented). In Mercator, the windowing metaphor as such has been dropped, while some important efforts have been carried out in

developing alternative means of non-visual interaction, such as the AudioRooms metaphor. However, there was still a design gap: the 3D-auditory interface provided interaction objects that were originally part of the object structure of visual applications; moreover, the plethora of so many auditory objects (as it is normal in 2D graphical interfaces), turned to be rather confusing for the blind user. This is due to the reduced human discrimination capability of concurrent auditory effects, even when their number is relatively small (e.g. less than 10). Such a problem is not attributed to the AudioRooms metaphor design as such, but rather to the fact that a visual graphical design, in which the user can normally cope with a large number of objects, is directly mapped to an audio structure. This remark highlights the importance of genuine user-oriented interface design, as opposed to automatic reproduction.

Other systems. Apart from the GUIB and Mercator projects, which have led to some spin-off commercially available products, there are examples of well known commercially available systems, which employ similar technical approaches, though providing less sophisticated non-visual dialogue capabilities; the following systems have been known to work for Windows 3.x: OUTSPOKEN™ by Berkeley Systems Inc. and SYSTEM 3™ by the TRACE Research & Development Centre at the University of Wisconsin, USA, for the Macintosh; SLIMWARE WINDOWS BRIDGE™ by Syntha-Voice Computers Inc., for WINDOWS 3.1™. A worth mentioning system for access by low-vision people is UnWindows [Kline et al, 1997], while other more recent systems working for Windows 95 are: ASAW™ (Automatic Screen Access for Windows), by MicroTalk [Asaw, 1997], and IN CUBE Enhanced Access™ (relying upon speech input and other commercial screen readers), by Command Corp Inc. [INCUBE, 1997].

1.4.1.2 Environment-level adaptations for motor-impaired users

In this context, motor impairment concerns the severe disability in using the standard means of input for graphical interfaces, such as keyboard and mouse. The most significant alternative access systems known for motor-impaired users are commercially available, and they follow a common practice: *provide alternative accessible input methods*. Only two of those systems will be briefly reviewed, which are considered in the context of this thesis as the most representative within this family of products. Both products provide methods to reproduce the input actions normally done via the keyboard and mouse, through appropriate switch-based press-actions from programmable superimposed graphical panels (being always “resident” on the display).

The main problem identified in such products is that interaction tends to be very slow. This is caused because various direct-manipulation interaction techniques in graphical applications, originally designed to be operated via the mouse (e.g. scrollbars, drag-and-drop), require a large amount of switch presses, in order to be performed via switches; for example, windowing management operations via switch-based reproduction of mouse actions is practically inefficient.

Switch Access to Windows (SAW™). This product, developed at the ACE Centre (UK), [SAW, 1992], has been available for WINDOWS 3.1, but it seems to be no longer supported for Windows 95 / NT. It relies upon the support of windows-based operations (e.g. focus control, window management, dialogue with interaction objects) via key bindings. The product offers a development kit to program such key bindings, on the basis of a scripting method, for creating a superimposed dialogue-box, on top of the windowing-environment, in which the programmed key bindings are represented as either graphical or textual elements. This dialogue box is made accessible via switches (i.e. sequential scanning); each time a particular item is selected, the corresponding key-shortcuts are sent to the particular listener application.

WinSCAN™. Developed by Academic Software Inc., “WinSCAN superimposes scanning control displays in front of application programs running under Windows on the PC. These control displays may be set up to contain words or pictures representing general functions to navigate Windows and/or specific functions to operate particular programs. The display can be set to scan at an adjustable rate vertically or horizontally and can be repositioned anywhere convenient on the PC desktop. Custom control displays can be created for particular programs, printed out, and saved on disk for later use or further editing.”, from [WinSCAN, 1997].

Other commercial systems. Apart from the previous two systems mentioned, there are numerous other commercial systems available, like: (i) WiVik™ [WiVik, 1997] family of on-screen keyboard programs for Windows, by Prentke Romich Company. (ii) Gus! Access Keyboard for Windows 95 [Gus, 1997], which allows single and dual switch users full access to Windows 95; in addition to offering text entry to any Windows application, it sets itself apart by incorporating a Scanning Windows 95 Task Bar, Scanning Start Menu and will even Scan the Menu Items (e.g. File Open etc) of any Windows application. (iii) CrossScanner™, “which provides a universal way to run all non-adapted software by single switch, or any pointing device”, [RJCooper, 1997]; and OnScreen™, “...on-screen keyboard allows the user to enter text into any application...gives you choices of words you've used previously, in the order of frequency-of- use, to complete your current word. It learns as you use it, adding to its built-in word bank.”, [RJCooper, 1997]; both are produced by RJ Cooper and Associates.

1.4.2 User-Adapted Interaction

1.4.2.1 Adaptive interaction.

Most of existing work regarding system-driven user-oriented adaptation, concerns the capability of an interactive system to dynamically detect certain user properties (i.e. during interaction), and accordingly decide various interface changes. This notion of adaptation falls in the adaptivity category, as defined in the context of this thesis, i.e.

adaptation performed after initiation of interaction, based on interaction monitoring information. It has been commonly referred to as adaptation during use. Although it has been considered as the best approach to suit user needs, there is always the risk of confusing the user with dynamic interface updates; this has led to the *hunting* concept [Browne et al, 1990a]: the system tries to adapt to the user, the user tries to adapt to the system; they will never reach a stable configuration. Of course, this is only a kind of philosophical argumentation, but identifies a potential problem.

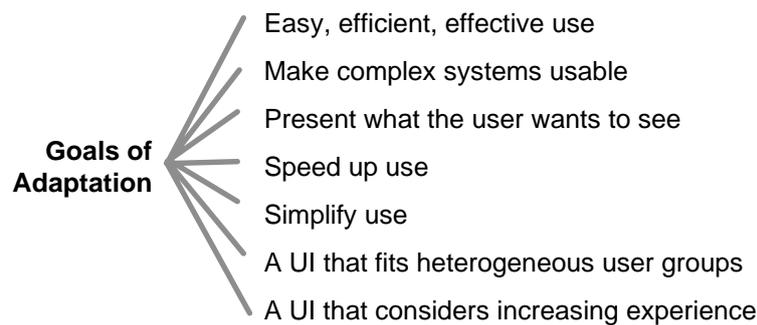


Figure 1.18 - Goals of adaptation, from [Dieterich et al, 1993].

“The main goal of adaptation in this respect is to present an interface to the user that is easy, efficient and effective to use”, [Dieterich et al, 1993]. Some of the goals of adaptation, as summarised in [Dieterich et al, 1993], are illustrated in Figure 1.18. It is evident from Figure 1.18, that the various adaptation goals have a very close relationship with the ISO 9241 guidelines for ergonomic design. However, those guidelines tend to be too general, defining properties to be asserted, rather than being prescriptive guidelines to accomplish ergonomic use, and mainly concern the interface design process. Hence, during the design of adaptation artefacts, those ergonomic principles should be certified (usability evaluation is likely to be involved), while in an adaptation process the goal is to match user attributes with the most suitable adaptation dialogue artefacts. In such a run-time decision making process, only if the engaged adaptation artefacts exhibit certified properties directly reflecting particular adaptation goals, then those general adaptation goals may constitute a measurable outcome of a run-time computation process. For instance, if an adaptation goal is to “minimise the possibility of errors”, then, for a particular end-

user and usage-context, the dialogue patterns provided to the user for the various tasks should be verified to support error minimisation; otherwise, with the absence of such metrics (i.e. goal satisfaction from various design artefacts), no meaningful matching of adaptation goals to design artefacts can be made during interaction.

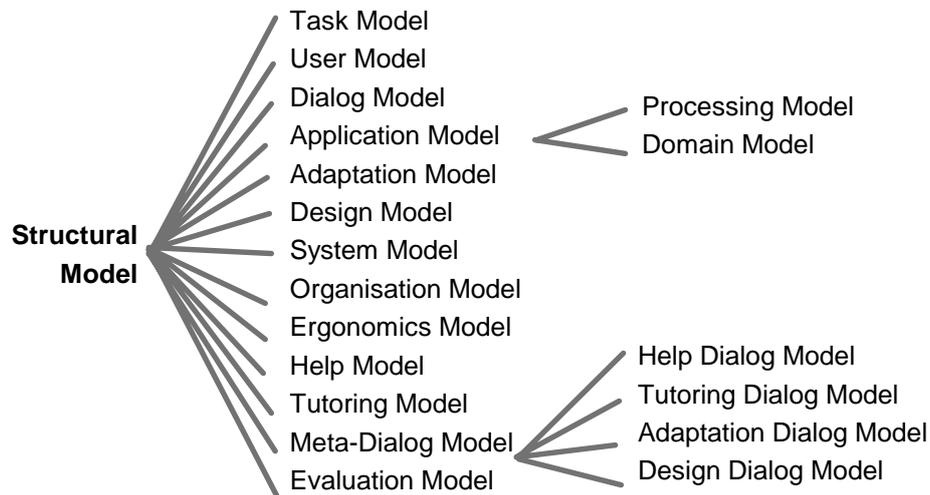


Figure 1.19 - Elements of the structural model for Adaptive User Interfaces, from [Dieterich et al, 1993].

1.4.2.1.1 Which Architecture ?

Although concrete software architectures are not clearly defined, there are various proposals as to what should fit, in a computable form, within an adaptive interactive system. In [Dieterich et al, 1993], all those categories of computable artefacts are summarised as being the necessary model categories within a structural model of adaptive interactive software (See Figure 1.19). In this summary, taken from [Dieterich et al, 1993], an appropriate reference to [Hartson et al, 1989], is made, regarding structure interface models:

“structural descriptive models of Human-Computer interface ... serve as frameworks for understanding the elements of interfaces and for guiding the dialogue developer in their construction”, from [Hartson et al, 1989].

However, developers require concrete software architectures in structuring and engineering interactive systems, and software systems in general. In this sense, the information conveyed within Figure 1.19 does not fulfil the requirements of an interface structural model, as defined in [Hartson et al, 1989], nor of a software architecture, as defined in [Jacobson et al, 1997] and [Thomas et al, 1995]. This fact leads to the initial argument that a concrete generic software architectural framework for adaptive interfaces, and automatically adapted interactions as a broader category, is completely lacking; this argument will be more supported and elaborated later on, as various aspects of previous work on adaptive interfaces are incrementally analysed.

1.4.2.1.2 User Models versus User Modelling Frameworks

In all types of adaptive systems, aiming to support user adaptivity in performing certain tasks, both embedded user models, as well as user-task models, played an important technical role. In [Kobsa et al, 1989], an important distinction is made between the user modelling component, encompassing methods to represent user-oriented information, and the particular user models as such, representing an instance of the knowledge framework for a typical user (i.e. individual user model), or user group (i.e. a stereotype model). But even this distinction still associates explicitly the user model with the modelling framework, thus establishing necessarily a dependency between the adaptation-targeted decision making software, which would need to process user models, with the overall user modelling component. This remark reveals the potential architectural hazard of making the system “too monolithic”, i.e. since the user model is linked directly with the modelling component, and decision making is associated with user models, all such knowledge categories should be physically located together.

1.4.2.1.3 Alternative Dialogue Patterns and Need of Abstraction

The need for explicit design, as well as run-time availability of design alternatives has been already identified in the context of interface adaptation [Browne et al, 1990b]. In view of the need for managing alternative patterns, the importance of abstractions has been identified, starting from the observation that design alternatives constructed with an adaptation perspective are likely to exhibit some common dialogue structures. In [Cockton, 1987] it is pointed out that “flexible abstractions for executable dialogue specifications” are a “necessary condition for the success of adaptable human-computer interfaces”. This argument implies that an important element in the recipe of success for making adaptive systems is to provide implemented mechanisms of abstraction in interactive software, allowing flexible run-time manipulation of implemented adaptation-oriented dialogue patterns. Although this important argument has been made a decade ago, the notion of abstraction has not been exploited by interface tools with explicit implementation support up-until now, with only a couple of exceptions [Wise et al, 1994], [Savidis et al, 1995a]. In the context of this thesis, the design and implementation of such abstraction mechanisms will be also discussed, as part of a development tool built to support unified interface construction.

1.4.2.1.4 Dynamic User Attribute Detection

The most common utilisation of internal dialogue representation has been for the collection and processing of interaction monitoring information. Such information, gathered at run-time, is internally analysed, by engaging some type of knowledge processing, to derive certain user attribute values (not known prior to initiation of interaction) which may drive appropriate interface adaptivity actions. A well known adaptive system employing such techniques is MONITOR [Benyon, 1984]. Similarly, for the purpose of dynamic detection of user attributes, a monitoring component in conjunction with a UIMS are employed in the AIDA system [Cote Munoz, 1993]. An important technical implication, due to the utilisation of dialogue models for deriving dynamic user attributes via interaction monitoring, is that dialogue modelling must be combined together with user models; thus, as discussed before, it inherently becomes associated with the user modelling component, as well as the adaptation-targeted decision making software. Effectively, this causes the overall adaptive system

architecture to “slide” more towards a monolithic structure, turning the development of adaptive interface systems to a more complicated task. It is argued that such an engineering complication, which potentially emerges when dynamic user attributes are to be detected, can be practically avoided.

More specifically, the problem is created due to an unnecessary software dependency, being the explicit association of the dynamic user attribute identification software with the dialogue model (i.e. interaction control implementation). Dialogue models for interactive systems need to be exhaustive, since they are computable models of the dialogue features of interactive applications. In order to derive dynamic user attributes, interaction monitoring information has to be collected and recorded by simply indicating the respective *dialogue context* in which it has been captured. Such recorded information constitutes a “montage” of the interaction history, and is usually matched against reliable action patterns (other techniques may be also employed), to detect particular user properties (e.g. confusion, preference, expertise, fatigue). At this point, the available dialogue related knowledge need not be in an executable form (as it would be needed for the dialogue control implementation), but merely in a representation structure allowing to convey the necessary knowledge pieces for the particular type of information / knowledge involved. This distinction is made since executable dialogue modes, like State Transition Diagrams [Jacob, 1988], State Charts [Wellner, 1989], Event Response Systems [Hill, 1986], Task Action Grammars [Payne, 1984], Propositional Production Systems [Olsen, 1990], etc., being the most representative in this category, tend to be very complex due to their linkage with the implementation world, as opposed to models primarily providing design-oriented representations, like task hierarchies [Johnson et al, 1988], GOMS notation [Card et al, 1983], etc. In conclusion, the technical proposition made is that *dialogue control software, and the dynamic user attribute detection software need not be tightly coupled.*

This argument is also supported by the fact that, in most available interactive applications, there is no such thing as an internal executable dialogue model other than programmed software modules; higher-order (i.e. relieving from the need of low-

level programming) executable dialogue models, as those previously mentioned, have been supported only by research-oriented UIMS tools, while, at present, the outcome of interface development environments, like VisualBasic™, is in a form more closely towards the implementation world, making the extraction of any design-oriented context rather impossible.

Hence, on the one hand, dynamic user attribute detection will have to necessarily engage dialogue-related information, while on the other hand, there is no chance that such required design information is practically made extractable from the interaction control implementation.

1.4.2.1.5 Interface Actions to Perform Adaptivity

The final step in a run-time adaptation process is the execution of the necessary interface updates at the software level. In this context, four categories of actions to be performed at the dialogue control level have been distinguished [Cockton, 1993], for the execution of adaptation decisions: (i) *enabling* (i.e. activation / deactivation of dialogue components); (ii) *switching* (i.e. selecting one from various alternative pre-configured components); (iii) *re-configuring* (i.e. modifying dialogue by using pre-defined components); and (iv) *editing* (i.e. no restrictions on the type of interface updates). The above categorisation seems to be mostly on the theoretical side, rather than reflecting an interface engineering perspective. Also, the term “component”, as used in the previous scheme, reflects mostly its old interpretation, i.e. relating to visual interface structures, rather than to its present establishment as referring to implemented sub-dialogues, including physical structure and / or interaction control. In this sense, it is argued that it suffices to define only two action classes, applicable on interface components: (a) *activate* components; and (ii) *cancel* activated components (i.e. deactivate). These two actions directly map to the implementation domain, (i.e. activation means “instantiation” of software objects, while cancellation means “destruction”), thus considerably downsizing the problem of modelling adaptation actions. This simple model is part of the unified engineering paradigm to be introduced and discussed in the context of this thesis.

1.4.2.1.6 Dimensions of User Models

User models are distinguished into three categories [Dieterich et al, 1993]: (a) *canonical* (i.e. “all users are the same”); (b) *stereotypical* (i.e. users belong to various categories; users falling in the same category share the same user model); and (c) *individual* (i.e. each user is different). Apart from this schema, there are other detailed aspects of user models requiring systematic analysis and classification. In all cases, user properties reflect a continuous decomposition of each feature category to various property categories, and recursively they derive sub-feature categories for each property category, leading to a hierarchical model structure, representing the user information space. Naturally, user models play the role of “input knowledge sources” in run-time adaptation-oriented decision making, while they may also be updated accordingly, due to dynamic user property detection. Even though existing adaptive systems employ user models for similar purposes, there have been diverse development strategies and implementation approaches, and, as mentioned in [Dieterich et al, 1993]:

“In the literature of adaptive User Interfaces, no architectural abstractions can be found. Mostly particular prototype architectures are reported, instead of abstract models [i.e. concrete generic architectures], while no clear pattern can be found between them.”

1.4.2.1.7 Structuring Dialogue Implementation for Adaptivity

The notion of *interface component* refers to implemented sub-dialogues provided by means of pre-packaged, directly deployable, software entities [Short, 1997]. Components increasingly become the basic building blocks in a component-based software assembly process, highly resembling the hardware design and manufacturing process. The need for configurable dialogue components has been identified in [Cockton, 1993], as a general capability of interactive software to visualise some important implementation parameters, through which flexible fine-tuning of interactive behaviours may be performed at run-time, still from within the software layer itself (i.e. self adaptation). However, the analysis made in [Cockton, 1993], is based on a theoretical ground, identifying mainly what is needed, without proposing how to approach this type of desirable functional behaviour; for instance, the distinction among “scalar”, “structured” and “higher-order” objects proposed does not map to any interface engineering practice. Moreover, the definition of adaptation policies as “changes” on different levels does not provide any concrete architectural model, nor reveals any useful implementation patterns. The results of such theoretical studies are good for understanding the various dynamics involved in adaptive interaction; however, they do not provide any added-value information for engineering adaptive interaction.

1.4.2.2 Adaptable interaction.

1.4.2.2.1 Software Tools for Lexical Adaptability.

Lexical adaptability is the automatic interface adaptation, prior to initiation of interaction, of lexical interface characteristics. This type of adaptation mainly refers to attributes of interaction objects (e.g. colour, font, position, background), as well as various other physical interaction properties, not directly related to object attributes (e.g. software control on mouse sensitivity, visual attributes of feedback techniques). The only system known to provide explicit support for lexical adaptability is the PIM tool [Savidis et al, 1997a]; this tool facilitates the establishment of alternative

programming layers on top of toolkits, while a distinctive property of the programming layers produced by the PIM tool is that they support a method for object attribute default values called Task-Based Attribute Default Method (TBADM). Attribute defaults complying to TBADM syntax, explicitly engage the user-task, together with object attribute specifications; this feature potentially allows design tools to generate directly such lexical attribute default values, since, there is a linkage with the design context (i.e. user task). In the context of the PIM tool, a lexical design assistant called USE-IT [Akoumianakis et al, 1996] has been employed to generate individualised configurations of object parameters; the programming toolkits generated by the PIM tool automatically parse and apply such individualised lexical attribute defaults, prior to initiation of interaction, thus performing lexical adaptability.

1.4.2.2.2 User-Model Based Lexical Design Assistants.

Such tools provide methods for modelling user attributes, and carry a decision making process which results in generated suggestions for lexical-level design. Those design recommendations directly engage physical dialogue constructs, like interaction objects and their respective attributes, as well as input / output devices to be selected. The only tool known to support this type of lexical design assistance is the USE-IT tool [Akoumianakis et al, 1996], which supports: (a) elicitation of the user model (includes user attributes as well as preferences); (b) elicitation of lexical level information (i.e. interaction objects and devices); (c) decision making, which results in lexical design recommendations.

1.4.3 User Interface Tools

1.4.3.1 Architectural Models of Interactive Systems

Various architectural models for interactive systems have been constructed in the past, having different technical objectives. Earliest work, following the appearance of graphical User Interfaces, concerned window managers, event mechanisms, notification-based architectures and toolkits of interaction objects. Quickly such architectural models have been supported by mainstream tools, thus they have been directly encapsulated within software interface technology; all available tools today support object hierarchies, event mechanisms and the call-back implementation model. There have been other architectural models, with a different focus, which, however, did not gain as much acceptance in the commercial arena, as it was originally expected. The *Seeheim* Model, and its successor the *Arch* Model, have been mainly defined to preserve the so called “principle of separation” between interactive- and non-interactive- code of interactive applications.

These models have been reflected primarily in research-oriented User Interface Management Systems [Myers, 1995]. Apart from these two architectural models, mainly referring to the *inter-layer* organisation aspects of interactive applications, there have been two other more implementation-oriented models, with an Object-Oriented flavour: the MVC Model and the PAC Model. Those models practically concern the *intra-layer* software organisation policies, by providing logical schemes for structuring the implementation code. All those four models, although mentioned as architectural frameworks in the past, are today considered as meta-models, since they do not meet the fundamental requirements of a software architecture, as defined by [Jacobson et al, 1997], according to which “an architecture should provide a structure, as well as the interfaces between components, by defining the exact patterns by which information is passed back and forth through these interfaces”.

1.4.3.1.1 The Seeheim Model

This model has been the first related to the run-time structure of interfaces produced by User Interface Management Systems (UIMS), and has reflected the orientation towards the physical and conceptual separation of the interaction-specific software from the non-interactive functional core. It introduces four logical system components (see Figure 1.20, left part): (i) *Application Interface Model*, which holds information to internally interface the functional core; (ii) *Dialogue Manager*, which holds information regarding the structure of dialogue and its syntactic rules; (iii) *I/O Manager*, which holds information about the structure and syntax of I/O items; and (iv) *Representation Manager*, which manages the mapping of I/O objects to internal objects, and vice versa. A discussion regarding the Seeheim model can be found in [Ten Hagen, 1990].

1.4.3.1.2 The Arch Model

This model has been introduced in [UIMS, 1992], and is considered to be the successor of the Seeheim model. It enhances the basic Seeheim model by introducing explicit interfacing layers among the various components. It has been called the Arch model because it was firstly drawn in a way geometrically resembling the Arch shape. It engages five logical components (see Figure 1.20, right part), three of which come from the Seeheim model (i.e. Dialogue Component, Domain Specific Component, and Interaction Toolkit Component), and two of which play the role of an interfacing layer (i.e. Domain Adaptor Component and Presentation Component), adapting, as required, the constructs, or the functionality provided by the components drawn below it (see Figure 1.20, right part). The Arch model made more explicit the need for *intermediate software layers* connecting the dialogue control implementation with both, the functional core, and the underlying toolkit.

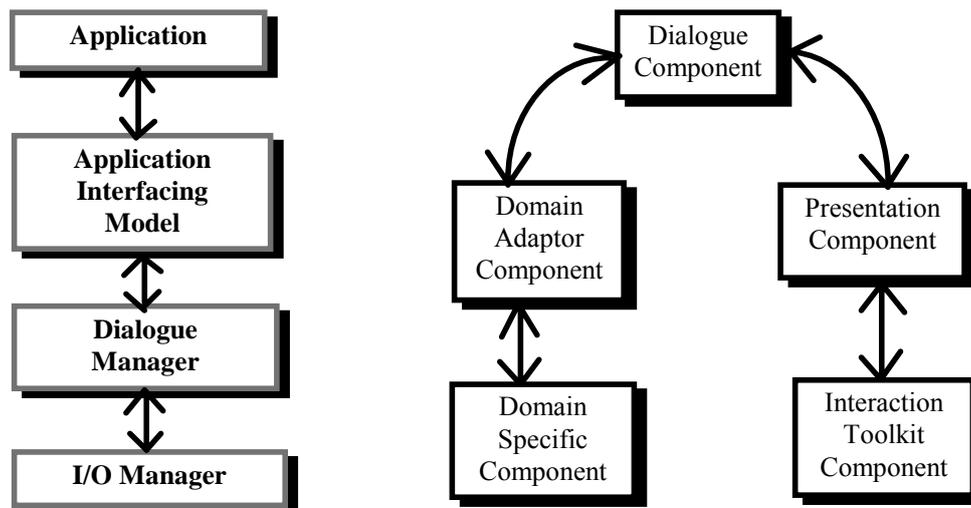


Figure 1.20 - The Seeheim model (left part), and its successor, the Arch model (right part).

1.4.3.1.3 The MVC Model

The Model-View-Controller (MVC) is an architectural model for building interactive systems which has been introduced in the context of Smalltalk-80 [Goldberg et al, 1984]. The essence of MVC is to invariably provide a clear separation between the internals of a system (functional core) and its corresponding interactive modules. Following the MVC model, an interface consists of three layers (see Figure 1.21): (a) *Model layer*, which implements the application functionality and provides the internal information structures; (b) *View layer*, which implements the mechanisms for presenting various aspects of the application layer to the user; and (c) *Controller layer*, which handles the user interaction with the application. The Controller processes user input, and invokes the appropriate Model function; when the work is done, the Model sends back a message to the Controller. The View updates the display in response to the Model messages, accessing also directly the model for further information. Thus the Model comprises a View and a Controller, but it never directly accesses any of them. The View and Controller on the other hand, access the Model's functions and the data when required.

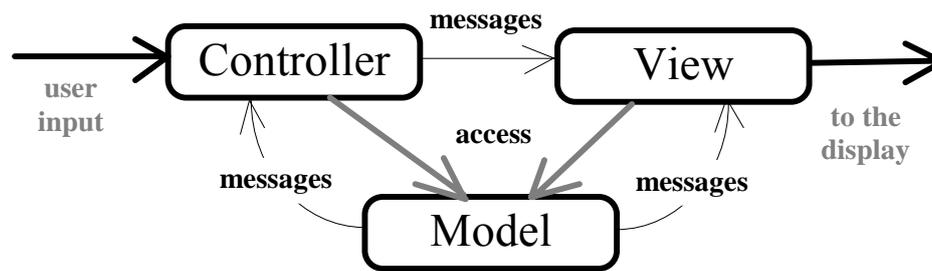


Figure 1.21 - The MVC model, from [Barkakati, 1991].

1.4.3.1.4 The PAC Model

The Presentation-Abstraction-Control model (PAC) recursively defines the organisation of interactive software as a hierarchy of PAC-agents. According to [Coutaz, 1990], “A PAC-agent defines competence at some level of abstraction. It is a three facet logical cluster which includes” (see Figure 1.22): (a) Presentation, that is a peceivable behaviour; (b) Abstraction, that is the functional core which implements some internal services and defines an interface to other agents; and (c) Control, that links Abstraction to a Presentation and maintains relationships with other agents.

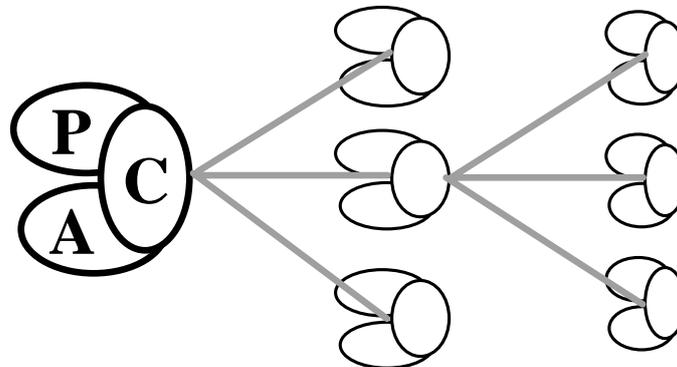


Figure 1.22 - The PAC model, supporting hierarchical organisation of agents by connecting to their respective control parts, from [Coutaz, 1990].

The C-part of an agent may communicate with corresponding C-parts of hierarchically higher or lower agents. A Presentation and its related Abstraction never communicate directly, but exchange data in their own formalism via a common Control. The difference with the MVC model is that I/O specific requests are

integrated in the Presentation component (in contrast to MVC, where this functionality is embedded in the Controller).

1.4.3.2 Tools for Building Accessible Interfaces

The concept of *design for all* and *universal access*, reflected in the “User Interfaces for All” objective in the context of Human-Computer Interaction, have been primarily defined to solve accessibility problems for disabled user groups. In this context, the top-level issue has been the design and implementation of interactive applications appropriate for the broader user population. However, before assembling directly accessible software products, it is required to employ, or even build from scratch, the appropriate interaction paradigms for disabled user groups. Instead, there have been mainly ad hoc adaptation approaches, realised with alternative access systems, which lacked a systematic design, evaluation and validation of the particular dialogue methods, as it was the case for the birth of the present interaction paradigms for able users.

There have been only a few efforts, mainly the outcome of funded research EC projects [GUIB Project, 1995], [ACCESS Project, 1996], which have led to some specialised tools for building interfaces for disabled users. In unified interface development, such tools provide interaction techniques which fall in the *resource dimension* of unified interfaces (see Figure 1.15, Section 1.3). In this context, some typical examples of diverse interaction elements which are to be manipulated for building unified interfaces will be presented

The tools which are briefly reviewed below are: (a) the COMONKIT Toolkit, supplying non-visual interaction elements complying to the Rooms metaphor; (b) the HAWK Toolkit, which provides flexible non-visual interaction elements through which various metaphoric representations can be realised; (c) the SCANLIB, which provides switch-based scanning interaction techniques on top of Windows, for motor-impaired user access; and (d) the HOMER UIMS, which is a tool for developing Dual

interfaces, being the first interaction paradigm in which the dialogues for able and disabled users are unified at both the design and implementation levels.

1.4.3.2.1 COMONKIT Non-visual Toolkit.

The COMONKIT Toolkit [Savidis et al, 1995b], has been developed in the context of the GUIB Project [GUIB Project, 1995], and included the design of a non-visual version of the Rooms metaphor. The principal requirements on which the design of the Rooms metaphor has been based are: (i) to provide a real-world spatial, but not necessarily visual metaphor, (ii) to enable blind users to conceive the interaction space in a manner similar to the way the real world-analogy is conceived in real life, and (iii) to facilitate easy memorization of the structure of the interaction space. The interaction space is structured in terms of Room objects, which form the main category of container-type entities. Rooms may enclose interactive entities (the type of which is defined via the realization phase). Room objects have *doors* and a *lift*; doors lead to other Room objects of the same floor, while the lift leads to Room objects which are either at a level above or below. The objects belonging to a Room object may be assigned to one of the following six parenthood groups: *front wall*, *back wall*, *left wall*, *right wall*, *floor* and *ceiling*.

Some of the various classes of interaction objects are: Menu, Textfield, Book (i.e. read only text reviewer), Switch (i.e. toggle), Button, Label, etc. In order to reproduce the "door" and "lift" concepts, it was decided to allow Room objects to be specified as children of Room objects, with the following representation to the user: in case that a child Room belongs to the vertical wall group, it is represented as a door leading to another Room object (which is the Room object itself), while in case that it belongs to the "floor" or "ceiling" groups it is made accessible through the lift which leads either to one level above (providing access to the Room objects of the "ceiling" group) or one level below (providing access to the Room objects of the "floor" group).

1.4.3.2.2 HAWK Non-visual Toolkit.

The HAWK Toolkit [Savidis et al, 1997c], has been developed in the context of the ACCESS Project [ACCESS Project, 1996], and aimed to provide flexible non-visual interaction entities supporting various metaphoric representations. It has been developed upon the experience gained by the COMONKIT Toolkit, in which the important role of container objects has been identified. In the HAWK toolkit, there is a single generic container class which does not provide any pre-designed interaction metaphor, but supplies appropriate presentation attributes through which alternative representations can be created. Also, the HAWK toolkit provides a comprehensive collection of conventional interaction objects, all of them directly supporting non-visual dialogue: Menu (exclusive choice selector object), List (multiple choice selector object), Button (push button analogy for direct command execution), Toggle (radio button analogy for on/off state control), Edit field (single line text input), Text reviewer (multi-line read-only text editor, with mark-up facilities). The non-visual navigation dialogue for the HAWK toolkit, allowing blind users to “move” among interaction objects (i.e. the analogy of the dialogue for sighted to navigate through windows and interaction objects), has employed advanced direct-manipulation facilities based on 3D-audio and 3D-pointing [Savidis et al, 1996].

The container class has the following attributes: (a) synthesized speech message (speech message to be given when the user "focuses" on the object); (b) Braille message (message displayed on Braille device when the user "focuses" on object); (c) "on-entry" digitised audio file (to be played when the user focuses the object); and (d) "on-exit" digitized audio file (to be played when the user "leaves" the object). This set of attributes enables each distinct container instance to be given a metaphoric substance by appropriately combining messages and sound-feedback. The HAWK toolkit has been utilized for building a non-visual electronic book [Petrie et al, 1996], as well as a non-visual interface for a Web browser [Stephanidis et al, 1997b].

1.4.3.2.3 SCANLIB Library for Switch-based Interaction.

The SCANLIB Library [Savidis et al, 1997b], has been developed in the context of the ACCESS Project [ACCESS Project, 1996], to facilitate the development of Windows 95 / NT interfaces, directly accessible by motor-impaired user groups. This effort has led to an augmented version of Windows object library with embedded switch-based scanning techniques, in which augmentation has been carried out at two levels: (a) at the interaction level, by augmenting all basic controls, including top-level windows and window management operations, so that dialogue with binary switches can be enabled; and (b) at the programming level, by visualizing new parameters to basic Windows controls (via inheritance on basic classes), in order to allow full implementation access for interface developers to all aspects of the augmented interaction techniques. Five different dialogue policies are implemented in SCANLIB, based on a classification of Windows objects into five meta-classes: top-level windows, container objects, text-entry objects, composite objects and button categories.

The scanning interaction techniques supported by SCANLIB are based on two fundamental actions: SELECT and NEXT, while on the basis of these two primitive actions, the dialogue for objects belonging to any of the five meta-classes has been designed and implemented. The SCANLIB library has been utilized to build an interpersonal communicator for speech-motor- and language-cognitive- impaired people [Kouroupetroglou et al, 1996], as well as an interface for a Web browser accessible by motor-impaired users [Stephanidis et al, 1997c].

1.4.3.2.4 HOMER UIMS for Dual Interface Development.

The HOMER UIMS [Savidis et al, 1998] has been developed in the context of the GUIB Project [GUIB Project, 1995]. It is a UIMS tool [Myers, 1995], supporting development through a specialised implementation language, called the HOMER language. This tool has been developed to support the development of Dual User Interfaces [Savidis et al, 1995a], a concept which aimed to introduce a new paradigm for solving accessibility problems for blind users. A Dual User Interface is

characterized by the following properties: (i) it is concurrently accessible by blind and sighted users; (ii) the visual and non-visual metaphors of interaction meet the specific needs of sighted and blind users respectively (hence, they may differ); (iii) the visual and non-visual syntactic and lexical structure meet the specific needs of sighted and blind users respectively (hence, they may differ); (iv) at any point in time, the same internal (semantic) functionality should be made accessible to both user groups through the visual and non-visual "facets" of the Dual User Interface; (v) at any point in time, the same semantic information should be made accessible through the visual and non-visual "facets" of the Dual User Interface.

It is evident that the concept of Dual Interfaces defines some properties which are very strongly related with the unified interface concept. Additionally, the HOMER UIMS introduced new development mechanisms, which are particularly useful when trying to marry together, in the implementation world, diverse interaction artefacts (like visual and non-visual interaction objects). Facilities for importing both visual and non-visual toolkits, as well as for constructing dual abstract interaction objects, playing a key role in dual interface development, have been made available for the first time within an interface tool. The HOMER UIMS has been utilised for building a dual picture exploration application, which provided interaction facilities for non-visual hierarchical exploration of annotated pictures [Savidis et al, 1995d].

Chapter II

UNIFIED USER INTERFACE DESIGN METHOD

2.1 THE NEED FOR POLYMORPHIC ARTEFACTS IN DESIGNING ADAPTED BEHAVIOURS

A variety of design approaches have been defined in the past, such as hierarchical task decomposition [Johnson et al, 1988] or GOMS analysis [Card et al, 1982]. The main purpose of the design information consolidated during the design process is to support subsequent development phases, like target implementation and / or usability evaluation. In this context, driven from the objective of automating or accelerating the transition between the design and implementation or evaluation phases, various alternative design approaches emerged, some of which have been provided with tool support for automatically realising the implementation and evaluation phases. Examples of such approaches are Task-Action Grammars (TAGs) [Payne, 1984], asynchronous models of user actions like CSP [Hoare, 1978], the User-Action Notation (UAN) [Hartson et al, 1990], propositional production systems (PPS) [Olsen, 1990], variants or hybrids of the original task model such as TADEUS [Stary, 1996], etc.

It is argued that new User Interface design methods emerge when there is a need to capture particular properties of interactive systems, which cannot be explicitly or sufficiently represented through existing design approaches. For instance, the identification of graphical constraints among interface objects is not normally realised in the context of a task analysis design process. Hence, if the explicit representation of graphical constraints is necessary (e.g. as an input to the implementation phase), a dedicated design process must be executed for the extraction of such necessary design information. In situations where a single design approach does not fulfil the requirements of the design process, the fusion of alternative techniques is applied, leading to hybrid design methodologies (e.g. combination of task analysis and graphic design methods).

Unified interfaces, which encapsulate automatically adapted behaviours, provide to end-users appropriate individualised interfaces instances. Hence, the process of designing unified interfaces should not lead to a single design outcome (i.e. a particular design instance for a particular end-user); rather, it has to collect and appropriately represent, all together, the various design “facets” that an automatically adapted interface may realise at run-time, for the various values of user- and usage-context- parameters.

But how do we organise all potential design instances ? Is it possible to explicitly produce the various alternative designs ? Producing and enumerating all distinct designs is not practically realistic, due to their large number, being related to all possible combinations of user- and usage-context- parameter values. The execution of such a large number of distinct design processes would be unacceptably resource demanding. Moreover, even if we assume that distinct designs can be practically produced, the implementation process would have to unavoidably turn all such design instances into independent interactive software systems; clearly, a wasteful software production strategy.

It is evident that running multiple design and implementation processes may not be a viable solution. Optimally, a single design process would be desirable, leading to a design outcome that may directly be mapped to a single software system implementation. In order to achieve this goal, the “hot” issue which had to be addressed was *how to organise the various design alternatives, emerging due to diverse user- and usage-context- parameters, into a single design structure.*

In this context, the first important remark is that design artefacts are produced to address particular design problems; alternative artefacts are required only when the particular design problems addressed impose diverse “solutions”, due to different user- and usage-context- parameters. The second remark is that design problems are tackled in a systematic manner in design techniques, starting from top-level problems, and, incrementally breaking down into smaller problems. Hence, there is a hierarchical pattern in the overall problem-solving and decision-making design

process, in which the particular nature of emerging design problems depends on the characteristics of the particular design technique; e.g. task identification for task-based design, graphical design for interactive design methods, input syntax definition for grammar-based design, etc.

Hence, two important properties are revealed: (a) the capability to associate alternative artefacts to a single design problem, depending on varying values of design parameters; this introduces the notion of a *polymorphic design artefact*, being a collection of alternative solutions for a single design problem, differing on the basis of user- and usage-context- parameter values. (b) the *hierarchical discipline* of the design process, in a “divide and conquer” strategy, in which design problems are incrementally broken-down and systematically addressed.

Existing design approaches support design processes which lead to a single artefact. The ability to differentiate and polymorphose is not embedded within available design techniques, thus requiring explicit enumeration of the alternative designs, in order to support representation of the numerous alternative interactive "facets" that a unified interface is capable to realise (due to the adaptation capability); this inability to unify alternative design patterns introduces technical problems when the resulting designs need to be implemented as a single interactive system. The unified interface design method has been defined in order to: (i) enable “fusion” of the potential distinct design versions into a single unified form, without, however, requiring the execution of multiple design phases; and (ii) produce a design structure which can be easily translated into a target implementation.

The unified interface design method builds upon the hierarchical task analysis method. It introduces the notion of polymorphic task decomposition, through which any task (or sub-task) may be decomposed in an arbitrary number of alternative sub-hierarchies [Savidis et al, 1997e]. The design process realizes an exhaustive hierarchical decomposition of various task categories, starting from the abstract level, by incrementally specializing in a polymorphic fashion (since different design alternatives are likely to be associated with differing user- and usage-context-

attribute values), towards the physical level of interaction. In a more compact definition, the unified design process drives an *abstract task definition with incremental polymorphic physical specialization*.

2.2 POLYMORPHIC TASK HIERARCHIES

The polymorphic task hierarchy decomposition method combines three fundamental properties: (a) *hierarchical organisation*, (b) *polymorphism*, and (c) *task operators*. The hierarchical decomposition adopts the original properties of hierarchical task analysis [Johnson et al, 1988] for incremental decomposition of user tasks to lower level actions. The polymorphism property provides the design differentiation capability at any level of the task hierarchy, according to particular user- and usage-context- attribute values. Finally, task operators, which are based on the powerful CSP language for describing the behaviour of reactive systems [Hoare, 1978], enable the expression of dialogue control flow formulas for task accomplishment; designers may freely employ additional operators as needed, while the basic set provided is illustrated in Figure 2.1.

Operator	Explanation
before	<i>sequencing</i>
or	<i>parallelism</i>
xor	<i>exclusive completion</i>
*	<i>simple repetition</i>
+	<i>absolute repetition</i>

Figure 2.1 - Basic task operators in the unified design method.

The concept of polymorphic task hierarchies is illustrated in Figure 2.2. Each alternative task decomposition is called a decomposition style, or simply a style, and is given an arbitrary name; the alternative task sub-hierarchies are attached to their

respective styles. The example polymorphic task hierarchy of Figure 2.2 shows how two alternative dialogue styles for a "Delete File" task can be designed: One exhibiting direct manipulation properties (i.e. "Direct Manipulation" style, file object is selected prior to operation to be applied), and another realising modal dialogue (i.e. "Modal dialogue" style, the delete operation is selected, followed by target file selection, followed by confirmation of operation).

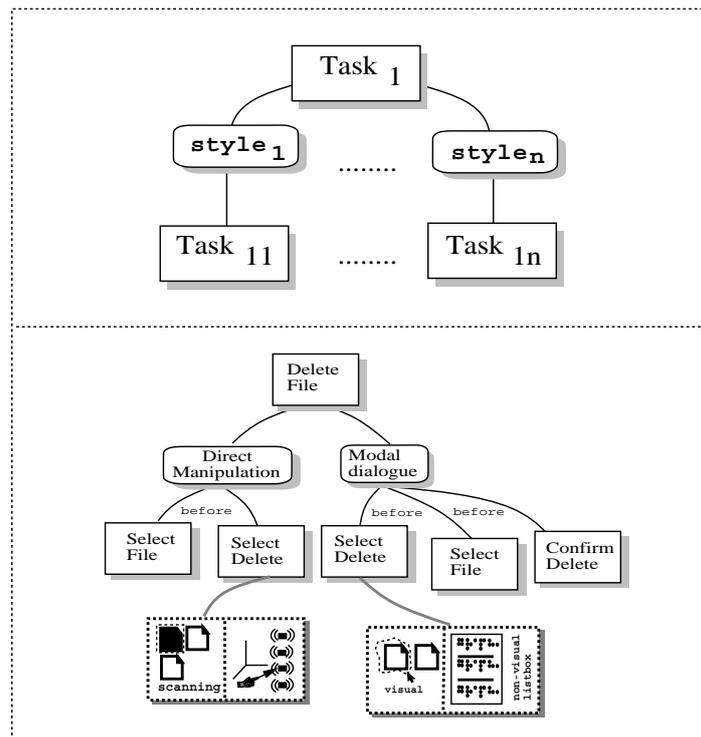


Figure 2.2 - The polymorphic task hierarchy concept, where alternative decomposition “styles” are supported (upper part), and an example polymorphic decomposition, for two diverse user groups (blind- and motor-impaired- users, lower part).

Additionally, the example demonstrates the case of physical specialisation. Since "selection" is an abstract task, it is possible to design alternative ways for physically instantiating the selection dialogue (see Figure 2.2, lower-part): via scanning techniques for motor-impaired users, via 3D hand-pointing on 3D-auditory cues for blind people, via enclosing areas for sighted able users, and via Braille output and keyboard input for deaf-blind users. The unified design method does not impose the designer to follow the polymorphic task decomposition all the way down the user-task

hierarchy, until primitive actions are met. At any level, a non-polymorphic task can be specialised following any design method chosen by the interface designer; for instance, in Figure 2.2 (lower-part) graphical illustrations are used to describe each of the alternative physical instantiations of the abstract "selection" task.

Similarly, the interface designer may choose a more suitable model for describing user actions for device-level interaction (e.g drawing, drag-and-drop, concurrent input) than the CSP- based operators which have been primarily chosen for expressing task relationships, such as an event-based representation (e.g. ERL [Hill, 1986], UAN [Hartson et al, 1989]).

2.2.1 Polymorphism versus Mutual Exclusion Operator

The most common questions regarding the need for a polymorphic task decomposition approach in the unified design method has been “why the representation of alternative sub-hierarchies by means of the traditional task-model, by employing the *xor* operator among alternatives, is not enough?”.

The answer to this question requires some elaboration (see also Figure 2.3). Firstly, the *xor* operator on the traditional task-model is interpreted as “the user is allowed to perform anyone, but only one, of the N sub-tasks”; this means that the physical context (i.e. interface components) for performing any of the N sub-tasks is made available to the user, while the user is required to accomplish only one of those sub-tasks. However, if alternative sub-hierarchies are related via polymorphism, it is implied that “a particular end-user will be provided with the one, and only one, of the N alternative designs, maximally matching the specific user attribute values”. Clearly, it is meaningless to provide all designed artefacts, which are likely to address diverse user attributes, concurrently to end-users, and force users to work with anyone of those (and only one); hence, the *xor* operator is not the appropriate way for organising alternative dialogue patterns.

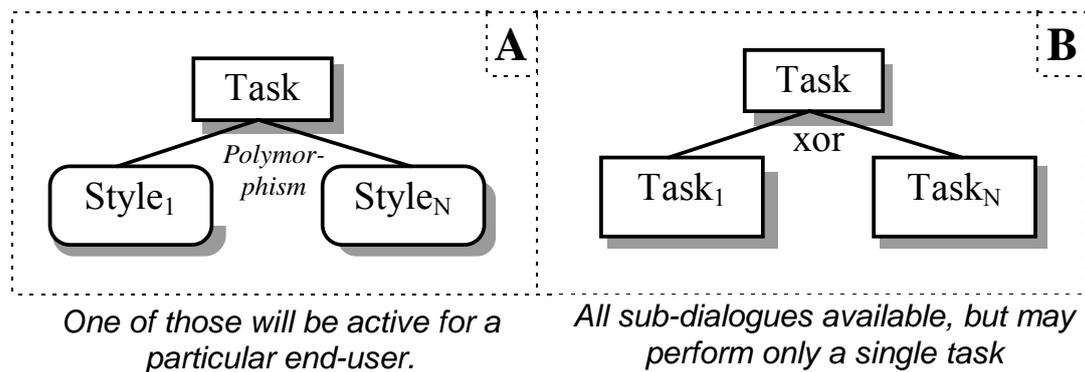


Figure 2.3 - (A) The use of polymorphism for alternative styles (i.e. dialogue patterns), when those address different user- and usage-context- attributes; and (B) the xor operator for various tasks, when exclusive completion has to be imposed.

As it will be discussed later in more detail, the design polymorphism mechanism entails a decision making capability for selecting among alternative artefacts, so as to assemble a suitable interface instance (regarding the end-user- and usage-context), while task operators support temporal relationships and access restrictions applied on the interactive facilities of a particular interface instance.

2.2.2 Polymorphism Entails Decision Making Capability for the Selection of Adaptation Artefacts

When a particular task is subject to polymorphism, alternative sub-hierarchies are designed, each associated to different user- and context- parameter values. A running interface, implementing such alternative artefacts, should encompass decision making capability, so that before initiating interaction with a particular end-user, the most appropriate of those artefacts are activated for all polymorphic tasks. Hence, polymorphism can be seen as a technique potentially increasing the number of alternative interface instances represented by a typical hierarchical task model. If polymorphism is not applied, a task model merely represents a single interface design instance, on which further run-time adaptation is restricted; in other words, *there is a*

fundamental link among adaptation capability and polymorphic design artefacts. This issue will be further clarified with the use of an example.

Consider the case where the design process reveals the necessity of having multiple alternative sub-dialogues available concurrently to the user, for performing a particular single task. This scenario is related to the notion of multimodality, which can be more specifically called *task-level multimodality*, in analogy to the notion of *multimodal input* which emphasizes pluralism at the input-device level. We will use the physical design artefact of Figure 2.4, which provides two alternative dialogue patterns for file management: one providing manipulation facilities, and another employing command-based dialogue.

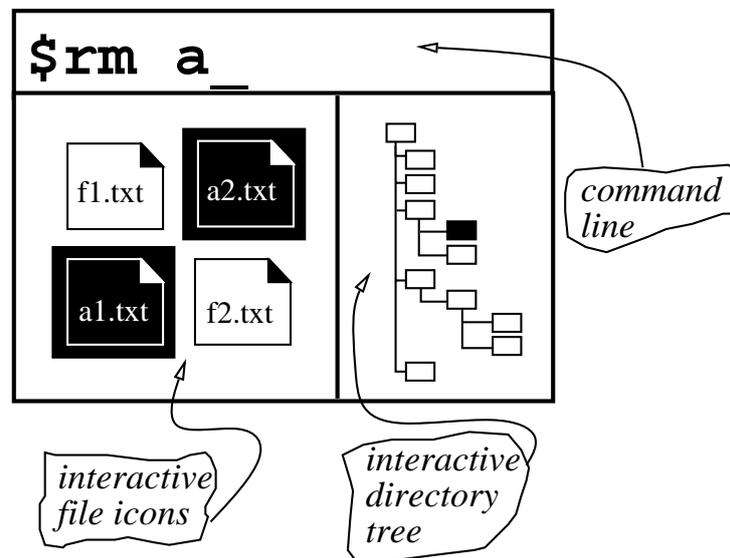


Figure 2.4 - A design scenario for alternative concurrent sub-dialogues, in order to perform a single task (i.e. task multimodality).

Both artefacts have been designed for the file management task, and can be represented as part of the task-based design, in two ways: (i) Through polymorphism, where each of the two dialogue artefacts is defined as distinct style; the two resulting styles are defined as being compatible, which this implies that they may co-exist at run-time (i.e. the end-user may freely use the command line or the interactive file manager interchangeably). (ii) Via decomposition, where the two artefacts are defined

to be concurrently available to the user, *within the same interface instance*, via the *or* operator; in this case, the interface design is “hard-coded”, representing a single interface instance, without needing further decision making. These two alternative approaches are illustrated in Figure 2.5.

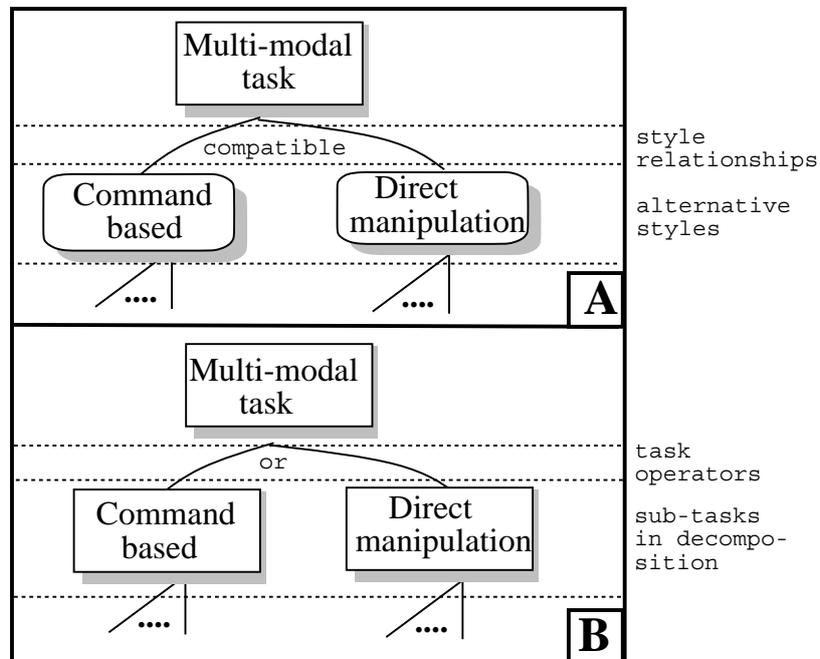


Figure 2.5 - Two ways for representing design alternatives when designing for task-level multimodality: (A) via polymorphism, adding run-time control on pattern activation; and (B) via or operator, hard-coding the two alternatives in a singular task implementation.

The advantages of the polymorphic approach are: (a) only one of the two artefacts may be decided to be present, depending on user parameters; (b) even if, initially, both artefacts are provided to end-users, if particular preference is dynamically detected for one of those, the alternative artefact can be dynamically disabled; and (c) if more alternative artefacts are designed for the same task, the polymorphic design is directly extensible, while the decomposition-based design will need to be turned to a polymorphic one, apart from the unlike case where it is still desirable to provide all defined sub-dialogues concurrently to the user.

2.2.3 Categories of Polymorphic Artefacts

In the unified design method there are three categories of design artefacts, all of which are subject to polymorphism on the basis of varying user- and usage-context-parameter values. These three categories, which are illustrated within Figure 2.6, are:

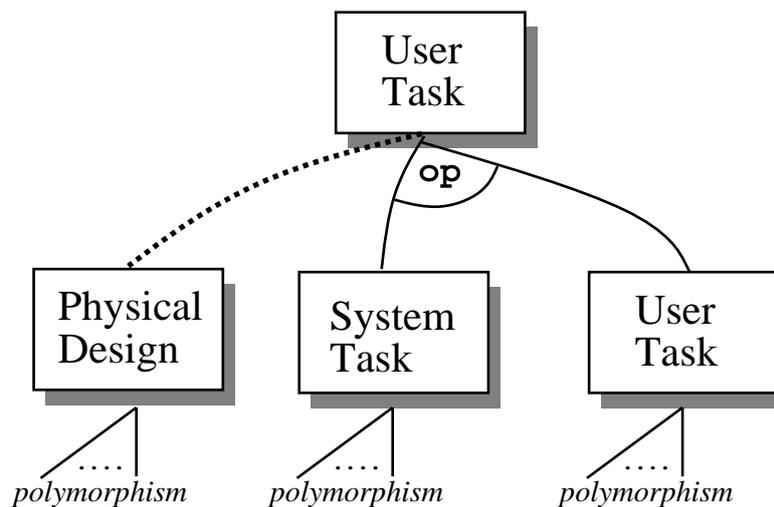


Figure 2.6 - The three artefact categories in the unified method, for which polymorphism may be applied, and how they relate to each other.

(a) *User tasks*, relating to what the user has to do; user tasks are the centre of the polymorphic task decomposition process. (b) *System tasks*, representing what the system has to do, or how it responds in particular user actions (i.e. feedback); in the polymorphic tasks decomposition process, they are treated in the same manner as user tasks. (c) *Physical design*, which concerns the various interface components (i.e. interface context) in which user actions are to be performed; physical structure may also be subject to polymorphism. System-tasks and user-tasks may be freely combined within task formulas, defining how sequences of user-initiated actions and system-driven actions are combined together. The physical design, providing the interaction context, is always associated to a particular user-task. It provides the physical dialogue pattern associated to a particular task-structure definition. Hence, it simply plays the role of annotating the task hierarchy with physical design information.

An example of such annotation is shown in Figure 2.2, where for the “Select Delete” task, the physical designs are explicitly drawn.

In some cases, given a particular user-task, there is a need for differentiated physical contexts, depending on user- and usage-context- parameters values. Hence, even though the same user actions are to be still performed (i.e. task decomposition is not affected), the physical design may have to be altered. One such representative example is relevant to changing particular graphical attributes, on the basis of ethnographic user attributes; for instance, in [Marcus, 1996], the choice of different iconic representations, background patterns, message visual structure, etc., on the basis of nationality information is discussed.

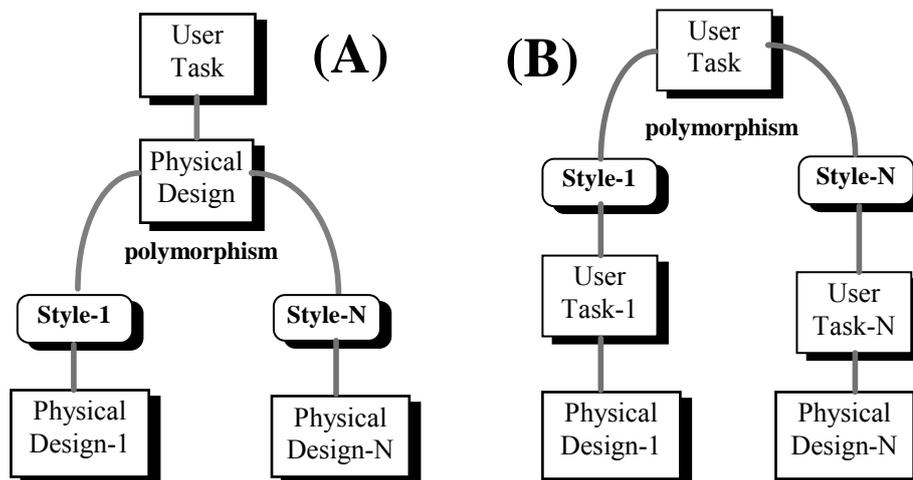


Figure 2.7 - Representation of alternative physical artefacts: (A) when for the same non-polymorphic task; and (B) when needed due to polymorphic task decomposition.

However, there also cases in which the alternative physical designs are dictated due to alternative task structures (i.e. polymorphic tasks); in such situations, each alternative physical design is directly attached to its respective alternative *style* (i.e. sub-hierarchy).

In summary, the rule for physical design artefact is: *if alternative designs are for the same task, then attach a polymorphic physical design artefact to this task, where the various alternative designs are associated as being the alternative styles of this polymorphic artefact* (see Figure 2.7, part A); *if alternative designs are due to alternative tasks structures (i.e. task-level polymorphism), attach each alternative physical design to its respective style* (see Figure 2.7, part B).

2.3 POLYMORPHIC TASK DECOMPOSITION PROCESS

User tasks, and in certain cases system tasks, need not always be related to physical interaction, but may represent *abstraction* on either user- or system- actions. For instance, if the user has to perform “selection”, then clearly, the physical means of performing the selection are not explicitly defined, unless the dialogue steps to perform selection are further decomposed. This notion of continuous refinement and hierarchical analysis, starting from higher-level abstract artefacts, and incrementally specialising towards to the physical level of interaction, is fundamental in the context of hierarchical behaviour analysis, either regarding tasks humans have to perform [Johnson et al, 1988], or when it concerns functional system design [Saldarini, 1989]. The “heart” of the unified design method is the *polymorphic* task decomposition process, which follows the methodology of abstract task definition and incremental specialisation, by additionally introducing the concept of polymorphic task decomposition, where tasks may be hierarchically analysed through various alternative schemes.

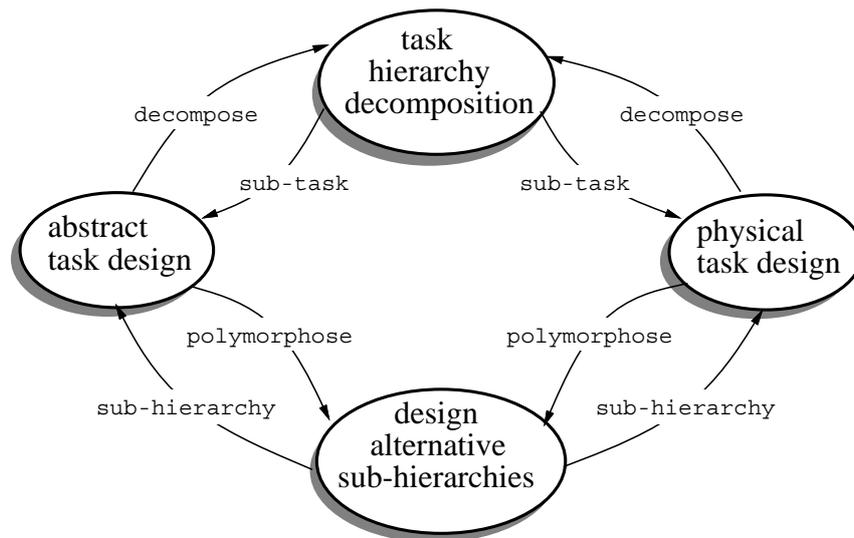


Figure 2.8 - The polymorphic task decomposition process in the unified design method.

It such a recursive process, involving tasks ranging from the abstract level, towards specific physical actions, decomposition is applied either in a traditional *unimorphic* fashion, or by means of alternative styles. The overall process is illustrated in Figure 2.8; the decomposition starts from abstract- or physical- task design, depending on whether top-level user tasks can be defined as being abstract or not. Next follows the description of the various transitions (i.e. design specialisation steps), from each of the four states illustrated in the process state diagram of Figure 2.8.

2.3.1 From “*abstract task design*” state

An abstract task can be decomposed either in a polymorphic fashion, if user- and usage-context- attribute values pose the necessity for alternative dialogue patterns, or in a traditional manner, following a unimorphic decomposition scheme. In the case of a unimorphic decomposition scheme, the transition is realised via a *decomposition* action, leading to the *task hierarchy decomposition* state. In the case of a polymorphic decomposition, the transition is realised via a *polymorphose* action, leading to the *design alternative sub-hierarchies* state.

2.3.2 From “*design alternative sub-hierarchies*” state

If this state has been reached, it means that various alternative dialogue styles required have been already identified, each initiating a distinct sub-hierarchy decomposition process. Hence, each such sub-hierarchy initiates its own polymorphic task decomposition process instance. While initiating each distinct process, we may either start from the *abstract task design* state, if the top-level task of the particular sub-hierarchy lays at an abstract level, or from the *physical task design* state, in case that the top-level task explicitly engages physical interaction issues. The two transitions both labelled as *sub-hierarchy*, from the *design alternative sub-hierarchies* state, are those leading to either of the two alternative process initiations for the decomposition of alternative sub-hierarchies.

2.3.3 From “*task hierarchy decomposition*” state

From this state, the sub-tasks identified need to be further decomposed. For each such sub-task, if it falls at an abstract level, there is a *sub-task* transition to the *abstract task design* state, else, if it explicitly engages physical interaction means, a *sub-task* transition is taken to the *physical task design* state.

2.3.4 From “*physical task design*” state

Physical tasks may be further decomposed either in a unimorphic fashion, or in a polymorphic fashion. These two alternative design possibilities are indicated by the *decompose* and *polymorphose* transitions respectively.

2.3.5 A Process Instance Example for Polymorphic Task Decomposition

The polymorphic task decomposition for the example of Figure 2.2 (lower part) will be briefly discussed, in view of the process model shown within Figure 2.8; the sequence of steps is illustrated under Figure 2.9 (states are mentioned with brief names). The initial state is *abstract task* (step 1), since "Delete File" can be defined as an abstract task. Through the *polymorphose* transition, two alternative styles are defined, resulting in two distinct sub-hierarchies (step 2).

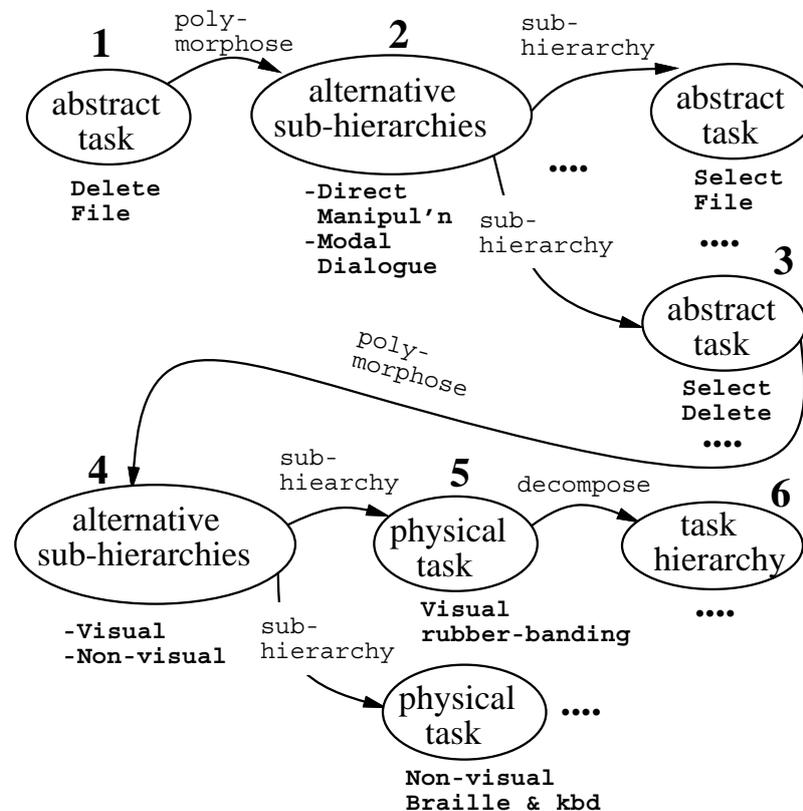


Figure 2.9 - An example of a polymorphic task decomposition process diagram.

Each such sub-hierarchy is further decomposed, by firstly deciding whether the top-level task is an abstract or a physical task, so as to continue the process (step 3). For instance, both "Select File" and "Select Delete" tasks defined are abstract (the rest are

not shown for clarity). Then, the steps for the "Select Delete" task are shown; this task is polymorphosed (step 4), resulting in two alternative sub-hierarchies (one for visual dialogue and another for non-visual dialogue). The top-level tasks for each of the two sub-hierarchies are in this case physical (step 5, as opposed to step 3, where all tasks are abstract). The "Visual rubber-banding" task is subsequently decomposed (step 6) to a unimorphic task hierarchy. It should be noted that alternatively to continuing the polymorphic task decomposition, we could alternatively continue from step 6, via any other appropriate design practice, such as event modelling.

2.4 DESIGNING ALTERNATIVE STYLES

The polymorphic task model provides the design structure for organising all the various alternative dialogue patterns of automatically adapted interfaces into a unified form. Such a hierarchical structure realises the fusion of all potential distinct designs which may be explicitly enumerated given a particular unified interface. Apart from the polymorphic organisation model, the following primary issues need to be also addressed: (i) when polymorphism should be applied; (ii) which are the user- and usage-context- attributes that need to be considered; (iii) which are the run-time relationships among alternative designed styles; and (iv) how the adaptation-rationale, connecting the designed styles with particular user- and usage-context- attribute values, is documented.

2.4.1 Identifying Levels of Potential Polymorphism

In the context of the unified design method, and as part of the polymorphic task decomposition process, designers should always assert that every decomposition step (i.e. those realised either via the *polymorphose* or through the *decompose* transitions of Figure 2.8) fulfils all the combinations of the target user- and usage-context- attribute values; these two classes of parameters together will be referred to as

decision parameters / attributes. More practically, in case that at any step of the task analysis process, there is a particular decomposition (for user- or system- tasks, as well as for physical design) which does not address some combinations of the decision attribute values, then all users and usage-contexts reflecting those attribute values have not been appropriately considered at this step of the design process. Such a design gap may be “closed” by constructing the necessary alternative sub-hierarchy(-ies) addressing the excluded decision attribute values, thus a level of potential polymorphism has been identified (i.e. need design alternatives).

2.4.2 Constructing the Space of Decision Parameters

We will discuss the definition of user attributes, being of primary importance, while the construction of context attributes may follow exactly the same representation approach. In the unified design method, the end-user representation technique as a finite set of attribute values [Savidis et al, 1997g] has been decided, because it does not pose restrictions on the employment of any particular user modelling approach for the implementation process. There is no predefined / fixed set of attribute categories. Some examples of attribute classes are: general computer-use expertise, domain-specific knowledge, role in an organisational context, motor abilities, sensory abilities, mental abilities, etc.

<i>Computer Knowledge</i>	expert	frequent	average	casual	naive	
<i>Web Knowledge</i>	very good	good	average	some	limited	none
<i>Ability to Use Left Hand</i>	perfect	good	some	limited	none	

Figure 2.10 - An example of a user-profile, as a collection of values from the value domains of user-attributes, from [Savidis et al, 1997g].

Also, the value domains for each attribute class are chosen by interface designers and human-factors experts as part of the design process (the value sets need not be finite). The broader the set of values, the higher the differentiation capability among various

individual end-users; for instance, commercial systems realising a single design for an "average" user have zero differentiation capability. The unified design method is not prescriptive with respect to which attribute categories to be chosen, as well as the target value domains of such attributes. Instead, it provides only the framework in which the role of user- and usage-context- attributes is explicitly positioned as part of the design process. It is the responsibility of interface designers to choose appropriate attribute categories and value domains, as well as to define appropriate design alternatives. A simple example of an individual user-profile, complying to the attribute / value scheme, is shown in Figure 2.10; for simplicity, designers may choose to collect in such profiles, only those attributes from which differentiated design decisions are likely to emerge.

2.4.3 Relationships Among Alternative Styles

The need for alternative styles emerges during the design process, when it is identified that some particular user- and / or usage-context- attribute values are not addressed by the various dialogue artefacts which have already been designed. Starting from this observation, one could argue that "all those alternative styles, for a particular polymorphic artefact, are mutually exclusive to each other"; in this context, exclusion means that, at run-time, only one of those styles may be "active".

Exclusion	<i>Relates many styles. Only one from the alternative styles may be present.</i>
Compatibility	<i>Relates many styles. Any of the alternative styles may be present.</i>
Substitution	<i>Relates two groups of styles together. When the second is made "active" at run-time, the first should be "deactivated".</i>
Augmentation	<i>Relates one style with a group of styles. On the presence of any style from the group at run-time, the single style may be also "activated".</i>

Figure 2.11 - Design relationships among alternative styles, and their run-time interpretation.

However, it is possible to design alternative styles addressing distinct decision parameter values, while still being meaningful to have some of the alternative artefacts concurrently together in the same adapted interface instance; for instance, in Figure 2.5 we have discussed how two alternative artefacts for file management tasks, a direct-manipulation and a command-based, can be both present at run-time. In the unified design method, four design relationships between alternative styles are distinguished (see Figure 2.11), defining whether alternative styles may be concurrently present at run-time. We will now show how these four fundamental relationships reflect pragmatic design scenarios.

2.4.3.1 Exclusion

The exclusion relationship is applied when the various alternative styles are proved to be only usable in the context of their target user- and usage-context- attribute values. For instance, assume that two alternative artefacts for a particular sub-task are being designed, aiming to address the “user expertise” attribute: one targeted to the “naïve” value, and the other targeted to the “expert” value. Then, these two are defined to be mutually exclusive to each other, since it is probably meaningless to have both dialogue patterns together at run-time (a “naïve” user would be potentially confused, an “expert” user would be provided with additional, practically redundant, interactive modules).

2.4.3.2 Compatibility

Compatibility is useful among alternative styles for which the concurrent presence during interaction, on one hand does not introduce usability problems, while on the other hand allows the user to perform certain actions in alternative ways. The most important application of compatibility is for *task-multimodality*, as it has been discussed in Section 2.2.2, where the design artefact of Figure 2.4 provides two alternative styles for interactive file management.

2.4.3.3 Substitution

Substitution has a very strong connection with adaptivity techniques. It is applied in cases that, during interaction, it is decided that some dialogue patterns need to be substituted by other. For instance, on the basis of monitoring during interaction the frequency-, as well as the pattern of use-, of certain operations, it may be decided to change both the ordering, as well as the arrangement of those operations; hence, the particular physical design style would need to be “cancelled”, while the appropriate alternative one would need to be “activated”. This sequence of actions, i.e. of a “cancellation” followed by an “activation”, is the realisation of substitution. In the general case, substitution involves two groups of styles: some styles are “cancelled”, being substituted by other styles which are “activated” afterwards.

2.4.3.4 Augmentation

This is better explained with an example. Assume that during interaction it is detected that the user is unable to perform a certain task, which leads to an adaptation decision (i.e. adaptivity action) in providing task-sensitive guidance to the user. Then, the interface component to provide such a feedback, exhibits the following properties: (a) it is not targeted in providing the interactive means to the user for performing that particular task, rather it empowers the user (by providing informative feedback) to accomplish a task more effectively; and (b) it does not affect the particular style(-s) already “active” at the point of “activation” (i.e. being compatible with them). In this case, the adaptive prompting dialogue pattern, which provides the task-oriented help, is related via an augmentation relationship with all alternative styles (of this specific task) with which it may be compatible at run-time.

2.4.4 Recording Design Documentation in Polymorphic Decomposition

During the polymorphic task decomposition process, there is a set of design parameters that need to be explicitly defined (i.e. given specific values) for each alternative sub-hierarchy defined. The aim is to capture, for each *sub-task*, the design logic for deciding possible alternative styles, by directly associating user-, usage-context- parameters and design goals with the constructed artefacts (i.e. styles). These parameters are: (i) *users & usage-contexts* (specific user- and usage-context- attribute values addressed by a style); (ii) *targets* (concrete design goals for a particular style); (iii) *properties* (the specific differentiating / distinctive interaction properties of a style, potentially in comparison to other styles); and (iv) *relationships* (how is a style related with other alternative styles). The values of these parameters are recorded during the decomposition process for each style in the form of a table. In Figure 2.12, an example is shown for the definition of these parameters regarding the two alternative styles for “Delete File” task (see Figure 2.2).

Task: Delete File	
Style: Direct Manipulation	Style: Modal Dialogue
Users & Contexts: Expert, Frequent, Average	Users & Contexts: Casual, Naïve.
Targets: Speed, naturalness, flexibility.	Targets: Safety, guided steps.
Properties: Object first, function next.	Properties: Function first, object next.
Relationships: Exclusion (with all).	Relationships: Exclusion (with all).

Figure 2.12 - An example of recording design documentation.

The purpose of design documentation is to capture design rationale associated to the various polymorphic artefacts; in this context, the notion of design rationale has a fundamentally different objective with respect to well known design space analysis methods. In the latter case, design rationale mainly represents argumentation about design alternatives and assessments [Bellotti, 1993] before reaching final design decisions, while in the our case, design rationale records the different user- and usage-

context- attributes, as well as design objectives underpinning the already made (i.e. final) design decisions. This set of five parameters previously defined, serves mostly as an "indexing" method for organising final design decisions with primary keys the particular *sub-task*, and the *users & usage-contexts* parameters. The outcome of the unified design approach is a single hierarchical artefact, composed of user- and usage-context- oriented final design decisions, associated to directly computable parameters (i.e. task, user- and usage-context- attributes). This unified structure, as it will be explained later on, is very close to an implementation-oriented software organisation model, thus, potentially constituting a valuable asset for the implementation phase.

2.5 ENGAGING ABSTRACT INTERACTION OBJECTS

During the task decomposition process, some sub-tasks can be directly related to user-input actions which can be managed via interaction objects. For instance, selecting from a list of options, interactively changing the state of a boolean parameter, providing an arithmetic value, etc., are all typical examples of input tasks which can be realised via the predefined dialogues implemented by various interaction objects. In such cases, it is desirable to employ general / abstract object classes, in order to enable alternative physical object classes to be selected, reflecting different user-, usage-context-, and domain- properties.

It is argued that designers primarily think in terms of specific instances and physical interface scenarios, especially if the task analysis and graphic design processes are carried out by different teams, rather than composing interface components via abstract behaviours and objects. In this context, we have defined a role-based model (see Figure 2.13) for "filtering" already made design decisions in order to identify "points" in which abstract interaction objects can be employed in the design representation. Three role categories for interaction objects are identified: lexical, syntactic and semantic; the description of each role follows.

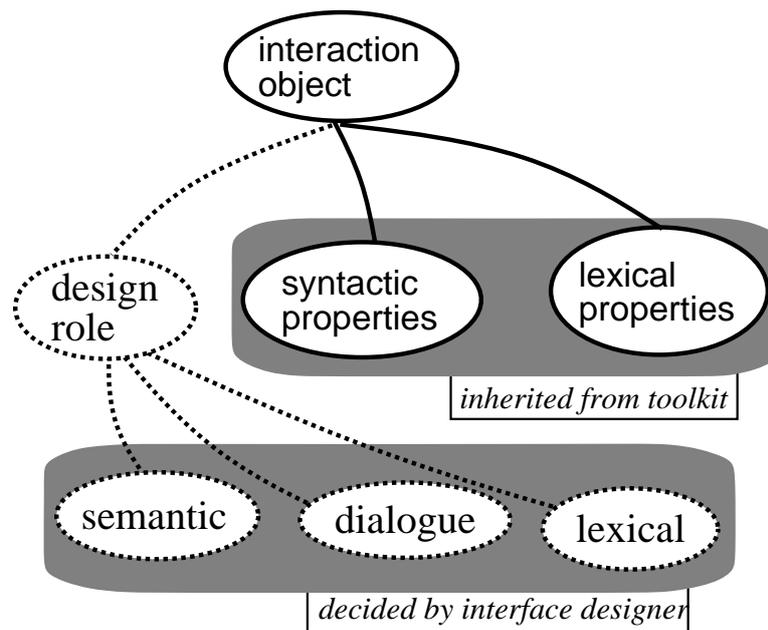


Figure 2.13 - Role-based model of interaction objects.

2.5.1 Lexical Role

In this case, the interaction object is employed for appearance / presentation needs. If such a role can be applied independently of physical realisation, then an abstraction can be identified. For example, assume a "Message" interaction object, which has only one attribute defining the message content (e.g. a string). It should be noted that the content could be verbal (i.e. the string is a phrase), if the user understands natural language, or even symbolic (i.e. the string is a file name where a symbolic sequence is stored), if the user understands symbolic languages. The presentation properties (e.g. emphasising with an icon, other visual / auditory effects) concern the physical implementation that may have alternative realisations.

2.5.2 Syntactic Role

The interaction object serves a specific purpose in the design of dialogue sequencing. If the role can be applied independently of physical realisation, then an abstraction can be identified. For example, a "continue" button, a "confirm" button, or a button to initiate an operation, all play the role of a "Command" given by the user, in the particular dialogue context. Such a "Command" class may be applied to provide, for instance, execution, confirmation, cancellation or progress, and may be applicable for various metaphors. It could be physically realised as a conventional push-button for the desk-top windowing interaction metaphor, as a voice-input command object for non-visual interaction, and as particular symbol structure for language-impaired users. The abstract interaction object "Command" may have only one boolean attribute to control whether it is accessible or not; the presentation feedback for indicating accessibility status, could be different, depending on its physical realisation.

2.5.3 Semantic Role

In this case, an interaction object interactively realises a domain object. For instance, an interaction object may present a domain object content, or provide the means to enable "editing" of the content by the user. In such cases, it is always possible to transform the role to a proper abstract class. A typical example is the provision of a numeric value by the user. A "Valuator" abstract object could be defined for this purpose, having various properties related to the type of numeric value required (e.g. range, discrete or real).

2.5.4 Re-engineering Designs Through the Role-Based Model

The role-based model can be applied on an existing physical design, in order to produce a higher-level design scenario. Such a scenario will serve as an abstract design representation, which may form the basis for deriving further alternative physical design scenarios. This notion of “filtering” physical scenarios via the role-based model, for subsequently constructing a higher-level design representation, is illustrated under Figure 2.14.

We will demonstrate the power of such design re-engineering process with a real life example. In Figure 2.15, a physical design artefact is shown, concerning a form-based dialogue, typically found in Web documents, for providing credit card information. In Figure 2.16, the physical design scenario is analysed in order to identify object roles, while in Figure 2.17, the resulting higher-level object model is outlined. Finally, under Figure 2.18 and Figure 2.19, two alternative physical design scenarios are produced, one for graphical interaction and another for non-visual Rooms interaction respectively, having the higher-level scenario of Figure 2.17 as the starting point.

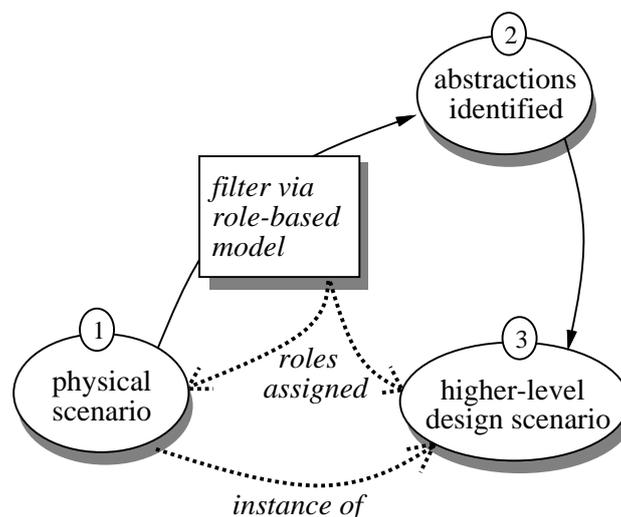


Figure 2.14 - Producing higher-level design scenarios through the role-based model.

Credit Card No: ^ _____
Expires: ^ __/____
 VISA MasterCard Access
 Other ^ _____
Submit

Figure 2.15 - The physical design scenario which will be re-engineered.

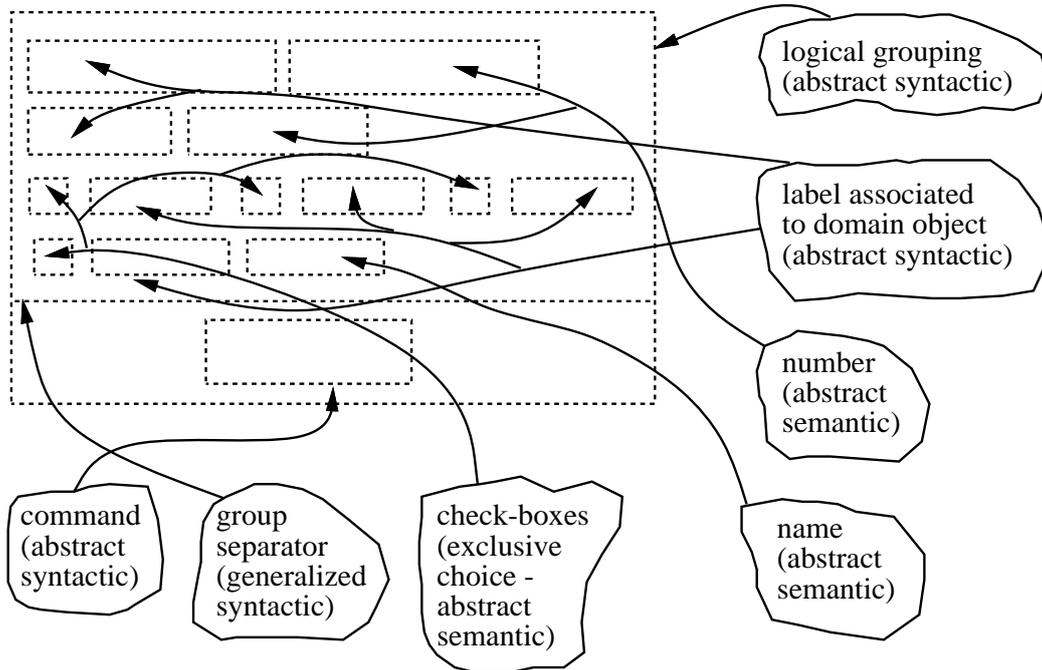


Figure 2.16 - Assigning roles to physical interaction objects.

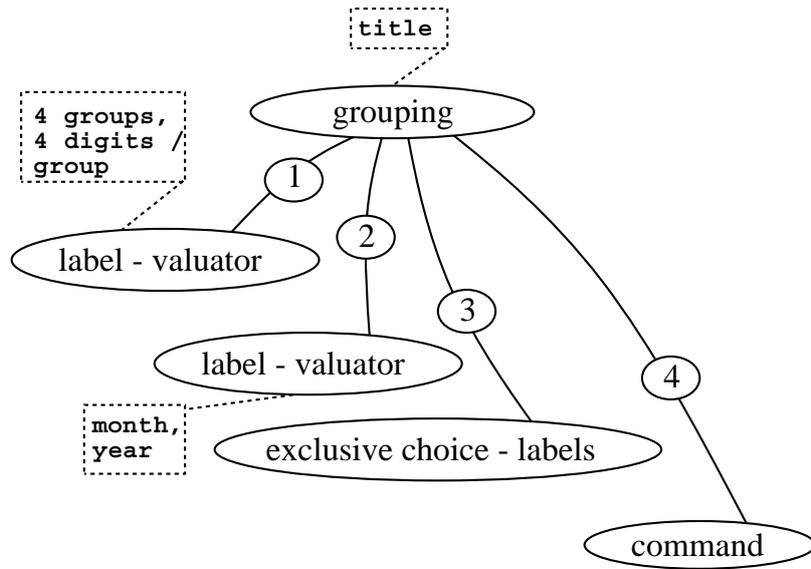


Figure 2.17 - The resulting higher-level object model.

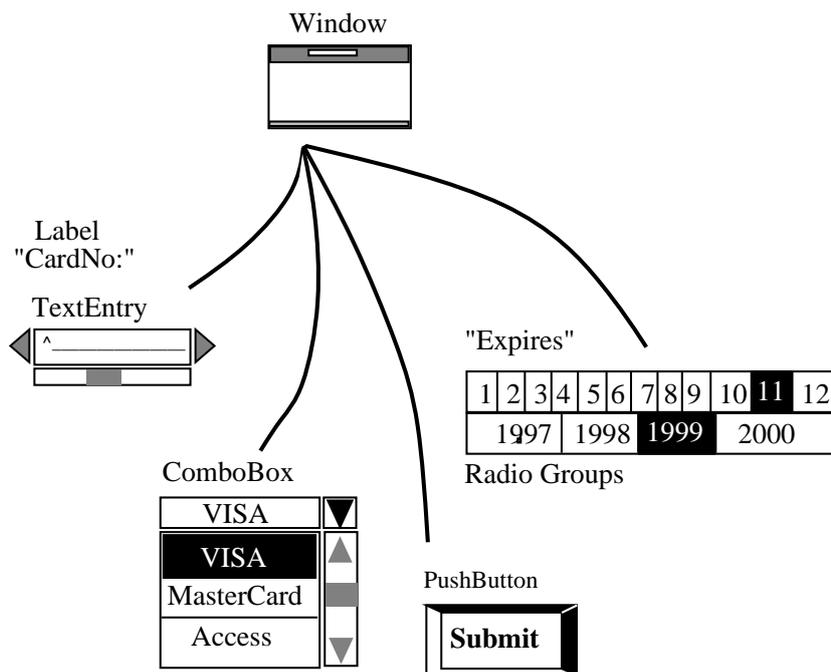


Figure 2.18 - An alternative graphical design derived on the basis of the abstract object model.

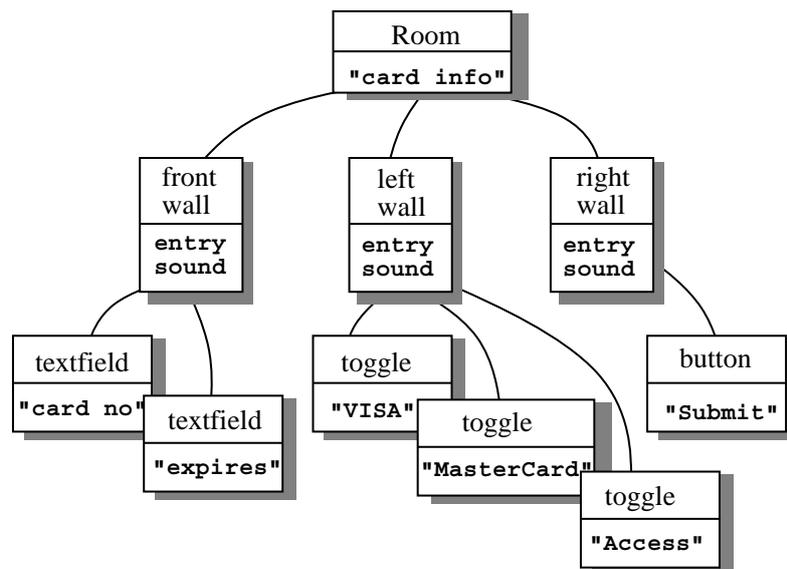


Figure 2.19 - An alternative non-visual Rooms design, derived from the higher-level object model.

2.6 INTERFACE DIFFERENTIATION EFFECTS FROM TASK-LEVEL POLYMORPHISM

During the unified design process, polymorphism may be applied at any level of the task hierarchy and for any task. The various behavioural / morphological differences which can be observed among the specific interface instances resulting from the alternative sub-hierarchies for a given polymorphic task, depend on the level of this task within the overall hierarchy. Such evident variations among alternative interfaces instances (due to the transformation behaviour) are characterised as differentiation effects. Three main categories of such interface differentiation effects are distinguished (see Figure 2.20), on the basis of the level at which polymorphism is applied (normally, combinations of these effects will be present).

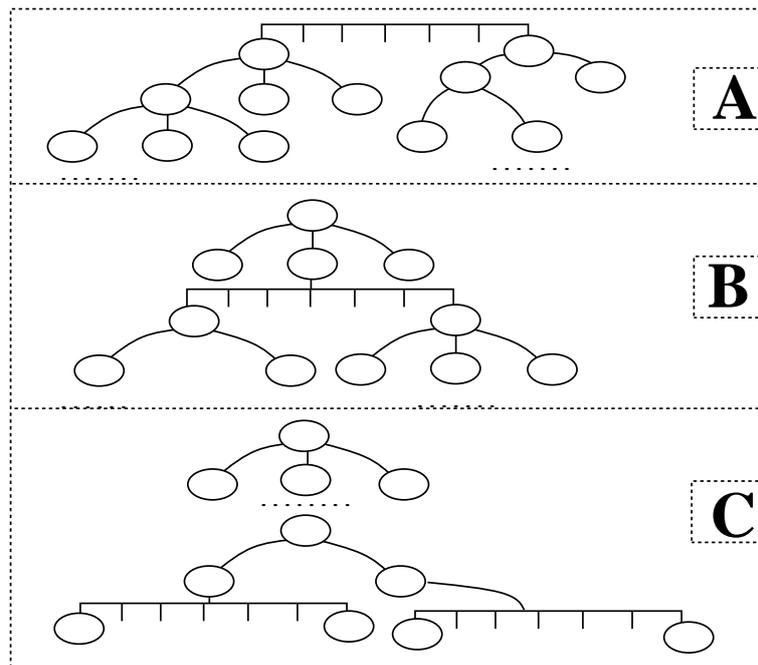


Figure 2.20 - Levels of task-based polymorphism: (A) for top-level tasks; (B) for intermediate tasks; and (C) for primitive tasks.

2.6.1 Polymorphism on *Top-level* Tasks

These tasks, which belong at the highest levels of the hierarchy and are called overall / main / top-level user tasks, concern what the user has to accomplish with an interactive application (see Figure 2.20, - part A); for instance, edit a document, send an e-mail, perform spell-checking, construct graphic illustrations, etc. When polymorphism is applied at this level, the interface instances are likely to be effected by structural differences, providing the feel of alternative versions of the same interactive environment (i.e. effect seems global).

2.6.2 Polymorphism on *Intermediate* Tasks

These tasks belong to the middle hierarchy levels, and concern particular dialogue states reachable after performing some required sets of actions (see Figure 2.20 - part B); for instance, dialogue boxes for setting parameters, executing selected operations, editing retrieved items, etc. The effect on alternative interface instances is to have overall similarities, with localised differences in interactive components and intermediate sub-dialogues.

2.6.3 Polymorphism on *Primitive* Tasks

These tasks appear at the lowest levels of the task hierarchy (leaves, see Figure 2.20 - part C) and concern primitive user actions (i.e. actions which can be directly supported by the primitive physical interaction elements); for instance, pressing a button, moving a slider, defining a stroke with the mouse, etc. The polymorphism at this level causes differences on device-level input syntax and / or on the type of interaction objects in some interface components.

2.7 DESIGN SCENARIOS FROM A PROJECT

The application of the unified interface design process will be discussed in the context AVANTI Project AC042 (Adaptable and Adaptive Interaction in Multimedia Telecommunications Applications), funded by the ACTS Programme of the European Commission (DG XIII). Some key design artefacts will be discussed, in situations where polymorphism has been applied, for the design of an adaptable and adaptive Web browser for HTML 3.2 [Stephanidis et al, 1997b]. The target user audience for the AVANTI project is: able-bodied, motor-impaired and blind users, with differing computer-use expertise, supporting use in various physical environments (office, home, public terminals at stations / airports, PDAs, etc). Some selected design

examples will be discussed, concerning specific tasks, through which the possible relationships among the alternative styles for a given polymorphic task will be revealed.

2.7.1 Link Selection Task

In existing Web browsers (e.g. Netscape Navigator™, Microsoft Explorer™ and SUN HotJava™), links in Web documents are activated by pressing the mouse left mouse button while the cursor resides within the area of a link. In Figure 2.21, the polymorphic design of the link selection dialogue (for textual links) is shown.

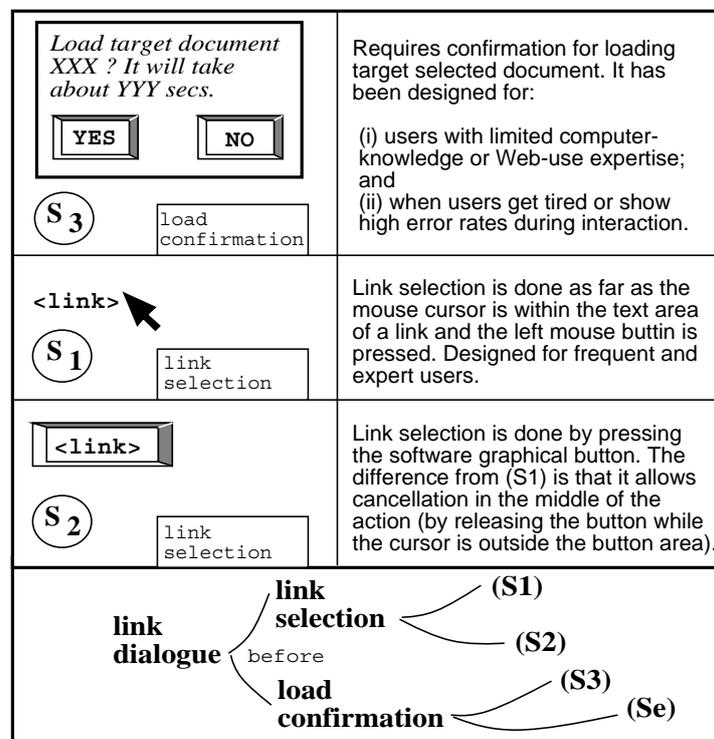


Figure 2.21 - Polymorphic decomposition of the *Link Selection* task.

There are two steps in link selection, according to the new design: (a) actually selecting a link (which can be done in two alternative ways, S₁ and S₂); and (b) requiring confirmation for loading target document (also done in two ways, S₃ and Se). The Se denotes an empty alternative sub-hierarchy (i.e. the task is considered to

be directly accomplished without any user actions). The design documentation for polymorphism is also presented in Figure 2.22 (only a brief summary is presented for clarity), while the relationships between the designed styles are indicated within Figure 2.22 (due to adaptivity, some dynamic style updates need to be explicitly mentioned).

<ul style="list-style-type: none"> • <i>S1</i> mutually exclusive with <i>S2</i>. • <i>S3</i> mutually exclusive with <i>Se</i>. 	<ul style="list-style-type: none"> • <i>S3</i> substitutes <i>Se</i> dynamically (if high error rates are detected).
<ul style="list-style-type: none"> • <i>Se</i> substitutes <i>S3</i> dynamically (if in a satisfactory interaction history, link activation has been always followed by positive confirmation). 	

Figure 2.22 - Relationships among alternative styles of *Link Selection* task.

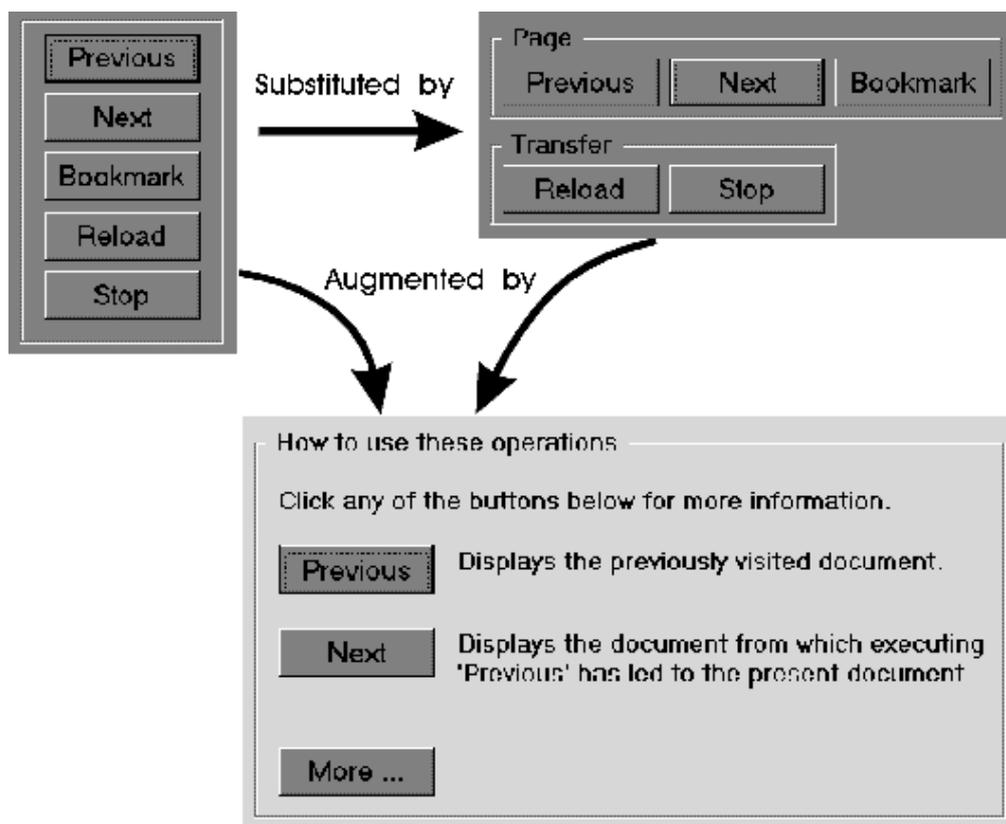


Figure 2.23 - Designed styles for the *Page Loading Control* task.

2.7.2 Document Loading Control Task

This task concerns typical operations that browsers provide to enable users control the Web page to be loaded and displayed (e.g. forward / backward, home, reload / stop, book-marking, load options). In Figure 2.23, two alternative style designs are shown, primarily designed for casual and naive users, which provide only a sub-set of the full range of operations (more advanced functions are only revealed to expert / frequent / average users). The relationships among the two styles are also indicated within Figure 2.24. Also, an adaptive prompting style is designed for the Page Loading Control task, aiming to help the user in performing this task, thus being an augmentation of the previous two alternative styles.

- | |
|---|
| <ul style="list-style-type: none">• The top-left style is the style to be initially active (“casual” or “naive” values on “application expertise” user attribute). |
| <ul style="list-style-type: none">• The top-right style is to dynamically substitute the previous style, in case that during monitoring it is observed that the user has used the operations successfully and became familiar with them. The new style groups logically operations with a title, and prepares the ground for the more advanced group called "options" to be included in future interaction sessions with this user. |
| <ul style="list-style-type: none">• The bottom style augments the particular active style, and it provides adaptive prompting / helping for carrying out the operations, in cases where inability to perform the task or high error rates are dynamically detected. |

Figure 2.24 - Relationships between designed styles for *Page Loading Control* task.

2.7.3 Page Browsing Task

The requirement to provide accessibility of the resulting browsers by motor-impaired users, necessitated the design of dialogues for enabling motor-impaired users to explore page contents and activate links. One globally applied technique has been to support hierarchical scanning of all objects in the interface via binary switches. In this case, motor impaired users can have access on the original visual interface designed for able users, though via alternative input techniques. This approach proved to be very good for all tasks, except from the case of page browsing; users spend a lot of time for switching between the scroll-bar (of the page presentation) and the visible

page contents, in order to identify desirable information / links. This has resulted in the two alternative styles, illustrated in Figure 2.25 and Figure 2.26, which augment the page presentation style, and are mutually exclusive. In Figure 2.25, the summary of links is presented in a window on the left of the Web page (i.e. all links collected and presented together).

In Figure 2.26, the document display on the left is automatically adjusted, so that the highlighted link is always visible; dashed arrows indicate that document context including the associated link is above / below visible portion, while solid arrows are attached at visible links and point to the exact link position in the document visible area; the number displayed above / below the scrollbar of the links' summary listbox indicates how many more links are included above / below the first / last displayed link within the listbox.

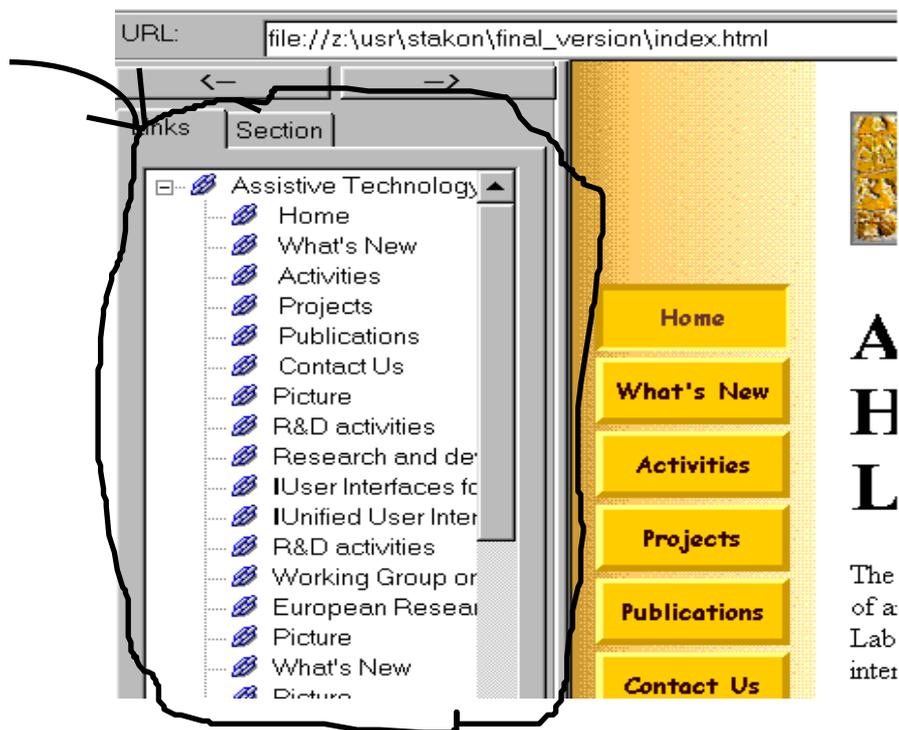


Figure 2.25 - The Link Summary style, version 1, for *Page Browsing* task.

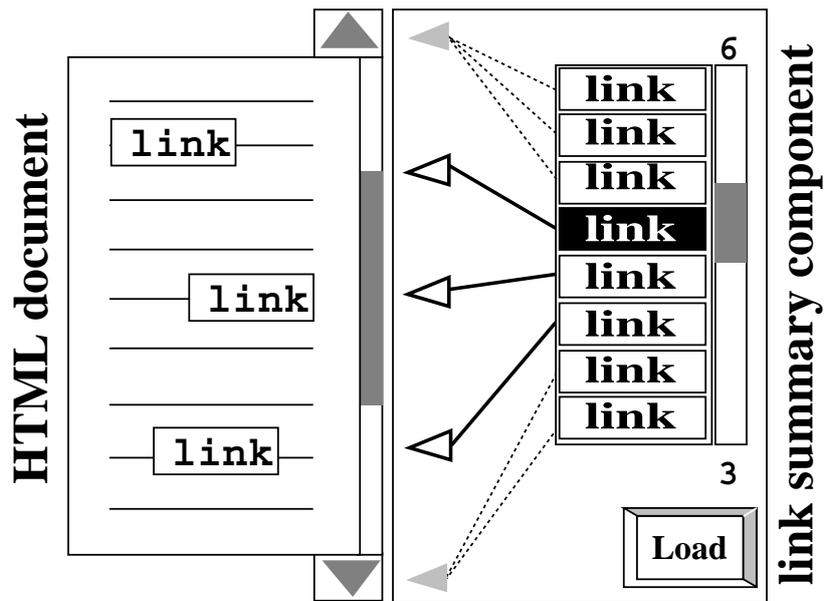


Figure 2.26 - The Link Summary style, version 2, for *Page Browsing* task.

2.8 DISCUSSION - UNIFIED DESIGN VERSUS DESIGN RATIONALE

The unified design method, as a design process, provides a methodology for organising design artefacts into a polymorphic unified structure. One of the fundamental properties of the unified design approach is its potential orthogonality with other particular design practices; in this context, we will discuss how the unified design approach can be combined with design rationale and design space analysis methods, into a fused process model.

Design rationale methods support argumentation about design alternatives and record the various design suggestions as well as their associated assessments. They are employed for the generation of design spaces which capture and integrate design information [Bellotti, 1993] from multiple sources, such as design discussions and diverse kinds of theoretical analyses. QOC (Questions, Options & Criteria) [McLean et al, 1995] design rationale is the most common method for constructing design

spaces, capturing argumentation about multiple possible solutions to design problems; QOC can be used to characterize the nature of contributions from diverse approaches, highlighting the strengths and weaknesses [Belloti, 1993].

It is clear that the employment of methods based on design rationale for constructing design spaces, leads to a comprehensive collection of alternative solutions (i.e. candidate design decisions), addressing their respective design problems, from which the most appropriate for their purpose will be finally chosen (i.e. final design decisions). Hence, possible alternatives are likely to exist during the design process; however, they are removed from the final design, since only the best choice (for each particular design problem) should be documented as an input to the implementation phase. In comparison, this is fundamentally different to the unified design method, where alternative styles directly represent final design decisions associated to particular user attributes.

We will show that the two approaches have different goals, and provide methodologies at different level of specialization; however, they may perfectly work in a complementary fashion, combining the benefits of both approaches. Firstly, we will analyze the scope of design space analysis for diverse design problems and tasks.

2.8.1 Design Space Analysis is a Meta-model for Cooperative Design Processes

The design space analysis method is very flexible with respect to the subject being studied. For instance, it may be initiated during a typical hierarchical task analysis method in order to create a design space for certain artefacts (e.g. choosing the proper metaphor for operations, deciding appropriate data visualizations, judging alternative visual interface designs). Also, we may turn the previous process up-side down, so that the subject of the design space analysis could be the formulation of the hierarchical task model; for example, making argumentation on the abstract task hierarchy and relevant parameters (e.g. task objectives, relationships, initiation conditions, feedback design).

In this sense, there is theoretically no restriction on how systematic, exhaustive, analytic and multidisciplinary a design space analysis process may become. It is argued that the justification of such an unlimited scope of design space analysis techniques is due to its nature of being a powerful design-process meta-model, rather than a specific design methodology. Any systematic methodology for cooperative interface design may be instantiated through the specific regulations and dynamics of design space analysis.

Moreover, it is believed that this meta-model is expressive enough to even manage the organization characteristics of the more general case of group-based decision making. This property enables design space analysis to be applied for virtually any problem domain; however, it should be used in conjunction with concrete and specialized design methods (many of which may be employed simultaneously within a single design project) when resolving specific design problems. This is one good reason why, in practice, design space analysis techniques have been mainly employed in the context of scenario-based design, where the key properties are argumentation, assessment and exchange of ideas over a collection of multiple concrete design artefacts serving a common purpose. If special purpose design practices are needed, such as hierarchical task analysis, design verification (e.g. performed via model-checking techniques), device-level interaction design (e.g. employing event-based notations), etc, the design space analysis provides only the framework for communicating design ideas, while it clearly does not provide the concrete design methodologies required.

2.8.2 Process-oriented versus Artefact-oriented Design Methodologies

As it has been previously discussed, the design space analysis technique constitutes a design process meta-model. Hence, it is primarily process-oriented and it gives particular emphasis on the organization of the design process, by means of a systematic communication and argumentation among designers, requiring well formulated and documented design suggestions.

On the other hand, the unified design method is primarily targeted on the production of finalized design artefacts into a form which is meaningful and mostly appropriate for the implementation of user-adapted interfaces. This is a fundamental difference between the two methods, which, as it will be explained later on, can be beneficially exploited through their combination into a comprehensive design procedure encompassing process-oriented and artefact-oriented properties. Both the design space analysis and the unified design method generate design spaces consisting of alternative dialogue patterns.

Is an explicit strategy provided for the construction of design alternatives ?	
UDM	Yes. The hierarchical polymorphic task analysis.
DSA	No. It is a meta-model for design processes.
Are the design alternatives produced considered finalised or under discussion ?	
UDM	All design alternatives are final design decisions.
DSA	All design alternatives included in a design space are mainly under discussion / argumentation.
What is the primary parameter giving birth to design alternatives ?	
UDM	Different user- and usage-context- attribute values.
DSA	Design criteria. It is also natural that new design ideas precede the identification of their respective criteria and properties.
How is the space of design alternatives related to the final design ?	
UDM	All design alternatives (i.e. styles) are part of the final design.
DSA	The final design is normally a small sub-set of the design space.
How are design alternatives related to each other ?	
UDM	Run-time relationships exist (exclusion, compatibility, substitution and augmentation).
DSA	They have an arbitrary number of design relationships (criteria-based).
What is the role of design rationale in the space of design alternatives ?	
UDM	It is a semantic indexing of the finalised design alternatives.
DSA	It semantically bridges non-finalised design alternatives and encompasses argumentation about each designed artefact.
How is the size of the design space documentation practically affected ?	
UDM	It is increased by considering new user- and usage-context- attribute values (i.e. polymorphism factor is increased).
DSA	Practically, the design space is enlarged if more designers are engaged in the design process.

Figure 2.27 - Key differences, concerning the life-cycle of design alternatives, between the Unified Design Method (UDM) and Design Space Analysis (DSA).

However, significant differences exist, such as the way design alternatives are generated, their relationship with respect to the final interface design, etc. We have

identified seven key issues which concern the life-cycle of design alternatives within each design technique. We show how these issues are addressed by the unified design method (UDM) and the design space analysis method (DSA) in a manner demonstrating their artefact-oriented and the process-oriented nature respectively (see Figure 2.27).

2.8.3 Fusing Unified Design and Design Space Analysis Methods

The design space analysis and the unified design method can be combined into a comprehensive design process. It is argued that any specialized design technique can benefit by being combined with the design space analysis approach. In our fused process model (see Figure 2.28), the overall design process is initiated by the polymorphic task analysis method. The *polymorphose* action of the unified design method will dictate the necessity for designing sub-hierarchies addressing specific user attribute values. In this context, such design decisions are analyzed and discussed by constructing appropriate design spaces.

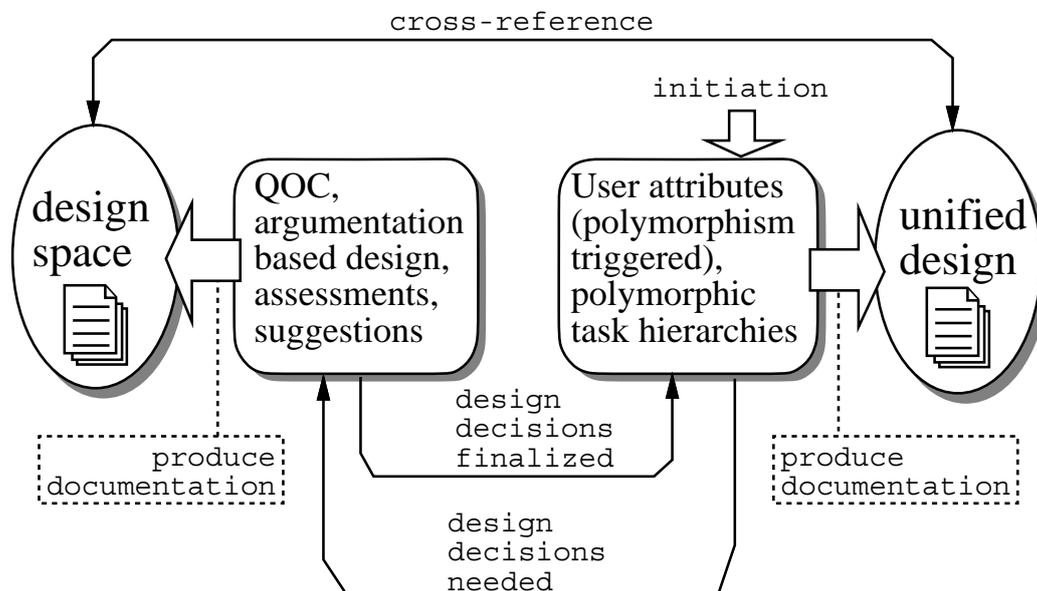


Figure 2.28 - The fused process model for combining the Unified Design Method with the Design Space Analysis method.

When final decisions are reached, they are fetched to the unified design sub-process, where they are recorded. Additionally, cross-references exist between the unified design documentation and the documentation of design spaces for: (i) elaborating design decisions (i.e. from the unified design documentation to design spaces' documentation); or (ii) summarizing design decisions (i.e. from design spaces' documentation to unified design documentation).

Chapter III

UNIFIED USER INTERFACE IMPLEMENTATION METHOD

3.1 UNIFIED INTERFACE SOFTWARE ARCHITECTURE

This Chapter will present a new software interface engineering approach, called *unified interface implementation method*, for accomplishing encapsulated adapted behaviour, which can be applied to any type of interactive software product or service. We emphasise the notion of *encapsulation*, which implies that, in order to realise a system-driven adaptation process, all the various parameters, decision making logic, and interface adaptation artefacts, are explicitly represented in a computable form, constituting integral parts of a running interactive system. The unified implementation method has been defined to be fully orthogonal with existing implementation techniques, without causing any inherent changes on the various specific design / implementation practices which are normally employed by the developers of interactive applications. As it will be shown, this distinctive property allows developers to apply the unified implementation approach upon existing interactive software (additional modules will still need to be introduced for coordination, control and communication purposes).

Central to the unified interface implementation method is the unified interface software architecture, which provides an architectural framework for structuring the interface implementation, encapsulating automatically adapted interactive behaviours. It should be noted that the architectural details which will be discussed comply with the definition of architecture [Thomas et al, 1995] from Object Management Group (OMG - the largest software consortium in the world), according to which an architecture should supply components, description of functional role per component, communication protocols among components or Application Programming Interfaces (APIs), as well as implementation and inter-operability issues. We will also briefly discuss some key issues concerning the effective marriage of the unified implementation method with prevalent software engineering practices, such as incremental development and re-usable dialogue patterns; we will also discuss in

more detail the blending of our approach with more recent development trends, like component-ware, as well as the Web evolution towards a layer for remote downloading and on-the-fly assembling of distributed interactive applications.

A unified interface performs run-time adaptation according to user- and context-attribute values (i.e. the decision parameters), thus practically providing different interface instances for different end-users and usage-contexts. We have previously introduced the most important functional properties of unified interfaces, namely, design knowledge representation and decision making, user- and context-representation, and dialogue pattern implementation.

In the elaboration of the unified interface architecture, apart from providing the various functional components, we will also demonstrate the inherent orthogonality of our approach with present prevalent interface architectural frameworks. In Figure 3.1, the components of the unified architecture are drawn as rounded rectangular shaded blocks on the vertical dimension, namely being: (a) Dialogue Patterns Component; (b) Decision Making Component; (c) User Information Server; and (d) Context Parameters Server. Additionally (see Figure 3.1), the components of a widely applied interface architecture (i.e. the *Arch meta-model* for interactive systems - [UIMS,1992]) are represented as rectangular greyed blocks on the horizontal dimension. The Dialogue Patterns Component is named Dialogue Control in the Arch meta-model, both playing a common functional role in their respective architectural framework; hence, even though two distinct blocks are drawn in Figure 3.1 for clarity, they are merged into a single black box, indicating that their architectural role is virtually identical.

As already mentioned, the unified architecture enables the deployment of existing interactive software, by requiring only some extra implemented modules, mainly serving coordination, control and communication purposes. In this context, as an instance of this capability, we will discuss the way in which the Dialogue Control module of a typical Arch-based interactive system may be easily expanded into a Dialogue Patterns Component, without affecting its original functional role. This

capability facilitates the vertical growth of existing interactive applications, following the unified architectural paradigm, so that automatically adapted interaction can be accomplished.

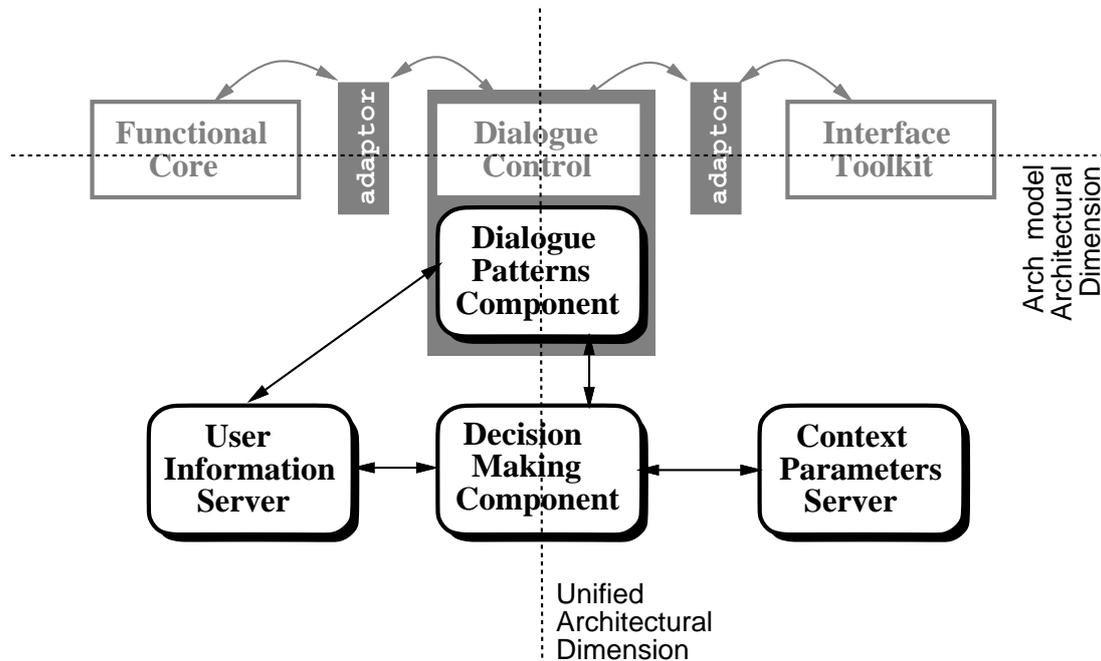


Figure 3.1 - The unified interface architecture (vertical dimension), being orthogonal with a typical run-time architecture of interactive systems (Arch meta-model, horizontal dimension).

The discussion on the unified architecture will be carried out as follows: Firstly, the functional role of each architectural component will be explained. Then, we will discuss the specific communication protocols among the various components, avoiding lower-level network implementation issues, while mainly focusing on the communication semantics. Finally, some important implementation issues will be drawn; in this context, we will also discuss the necessary development steps for vertically expanding existing interactive software to play the role of a Dialogue Patterns Component.

3.2 DESCRIPTION OF COMPONENT FUNCTIONAL ROLES

3.2.1 User Information Server

This module provides access to the individual profiles of end-users, which, together with usage-context information, constitute the primary input source for the adaptation decision making process (taking place in the Decision Making Component). An initial end-user supplied identification (i.e. some kind of password upon start-up) is required by the User Information Server, for retrieving the corresponding user profile. In the case of desk-top systems with dedicated users, all interactive applications may gain a hard-wired "user id", otherwise explicit user identification is needed. Internally, user profiles may be modelled in any appropriate form, depending on the type of user-oriented information that the User Information Server should feed to the decision making process.

For instance, it may employ other additional knowledge-based components for processing retrieved user profiles, making additional assumptions about the user, or even updating the original user profile attribute values; systems like BGP-MS [Kobsa, 1990], PROTUM [Vergara, 1994], or USE-IT [Akoumianakis et al, 1997] may be employed for such intelligent processing purposes. Apart from such initial (i.e. prior to initiation of interaction) manipulation of user profiles, it may also collect and process at run-time interaction monitoring information, in order to draw additional inferences about the end-user. Such inferences may result in identification of: user preferences, loss of orientation in performing certain tasks, fatigue, inability to complete a task, etc. In the communication with the rest of the architectural components, the user-oriented information is conveyed as a sequence of attribute values.

3.2.2 Context Parameters Server

This component encompasses information regarding usage environment and interaction-relevant machine parameters. During the interface design process, the identification of those important parameters, relevant to the context(-s) of use, will need to be carried out. This module is not intended to support device independence, but to provide *device awareness*, thus enabling the Decision Making Component to select those interaction patterns, which, apart from fitting the particular end-user attributes, are also mostly appropriate for the type of the equipment available on the end-user machine. The usage-context attributes values are communicated in the form of value sequences to the Decision Making Component, before initiation of interaction; additionally, during interaction, some dynamically changing usage-context parameter values will be also fed to the Decision Making Component for *adaptive* decisions. For instance, assume that the initial decision for feedback method is to employ audio effects; then, the dynamic detection of an increase in environment noise rate may result in deciding, during interaction, to switch in a visual feedback approach.

3.2.3 Decision Making Component

This module encompasses the logic for deciding the necessary adaptation actions, on the basis of the user- and context- attribute values received from the User Information Server and the Context Parameters Server respectively. Such attribute values will be supplied to the Decision making Component, prior to initiation of interaction (i.e. initial values, resulting in initial interface adaptation - *adaptability*), as well as during interaction (i.e. changes on particular values, resulting in dynamic interface adaptations - *adaptivity*). The Decision Making Component is responsible to only decide the necessary adaptation actions, which are then directly communicated to-, and subsequently performed by-, the Dialogue Patterns Component.

As part of the unified interface design process, alternative dialogue design artefacts may needed to be designed, even for various artefact categories (i.e. user tasks, systems tasks and physical design), due to different values of the user- and usage-context- parameters. In the implementation process, all these distinct dialogue artefacts become implemented interactive components. The run-time adaptation process, for a particular end-user and usage-context, is practically a selection process, from the “pool” of alternatives implemented interaction components (residing in the Dialogue Patterns Component), of those components which have been designed to address that particular end-user and usage-context attribute values.

	Adaptability	Adaptivity
<i>Initial selection of UI components</i>	<ul style="list-style-type: none"> • Via <i>activation</i> actions. 	<ul style="list-style-type: none"> • <i>Not for adaptivity.</i>
<i>Dynamic selection of UI components</i>	<ul style="list-style-type: none"> • <i>Not for adaptability.</i> 	<ul style="list-style-type: none"> • Via <i>activation</i> actions.
<i>Dynamic cancellation of UI components</i>	<ul style="list-style-type: none"> • <i>Not for adaptability.</i> 	<ul style="list-style-type: none"> • Via <i>cancellation</i> actions.
<i>Dynamic substitution of UI components</i>	<ul style="list-style-type: none"> • <i>Not for adaptability.</i> 	<ul style="list-style-type: none"> • Via <i>cancellation</i> actions, followed by the necessary <i>activation</i> actions.

Figure 3.2 - The User Interface (UI) component manipulation requirements (left), for either *adaptability* or *adaptivity*, and their expression via *cancellation / activation* adaptation actions.

In order to perform such a resolution process, the Decision Making Component encompasses information regarding all the various dialogue patterns collected within the Dialogue Patterns Component, and their specific design role (to serve particular values of user- and usage-context parameters). There are two categories of adaptation actions which are decided and communicated to the Dialogue Patterns Component: (i) *activation* of specific dialogue components; (ii) *cancellation* of previously activated dialogue components. These two categories of adaptation actions are adequate to express the various interface component manipulation requirements for realising either *adaptability* or *adaptivity*, as they have been defined under Section 1.2.3 (see Figure 3.2).

3.2.4 Dialogue Patterns Component

This module implements all the various alternative dialogue patterns identified during the design process, on the basis of various user- and context- attribute values. The Dialogue Patterns Component may employ pre-developed interactive software, in accordance to additional interactive components. The latter case is normally expected, if the directly deployed interactive software does not provide all the required implemented patterns, addressing the target user- and usage-context- attribute values, for accomplishing adapted behaviours.

The Dialogue Patterns Component should be capable to apply pattern activation / cancellation decisions originated from the Decision Making Component. Additionally, interaction monitoring components may also be attached to various implemented dialogue patterns, providing monitoring information to the User Information Server for further processing (e.g. key-strokes, notifications for use of interaction objects, task-level monitoring). The particular level- and frequency- of monitoring are to be requested at run-time by the User Information Server.

3.3 INTER-COMPONENT COMMUNICATION IN PERFORMING ADAPTATION

The final stage of an *adaptation cycle* is performed when the necessary interactive adaptation artefacts become available to the end-user. Since adaptation may take place either once, prior to initiation of interaction (i.e. *adaptability* oriented), or an arbitrary number of times, after initiation of interaction (i.e. *adaptivity* oriented), it is clear that multiple adaptation cycles may be potentially performed. Irrespective of the processing layers involved “behind the scenes”, the most important outcome of an adaptation cycle are the decisions for activation / cancellation of interface components; those are generated by the Decision Making Component, and are then

subsequently applied (leading to the completion of an adaptation cycle) by the Dialogue Patterns Component.

3.3.1 *Adaptability and Adaptivity Cycles*

The completion of an adaptation cycle, being either adaptability- or adaptivity-oriented, is realised in a specific number of distributed processing stages, performed by the various components of the unified architecture. During those stages, the components communicate to each other, requesting or reporting specific pieces of information. The overall communication requirements among the components are illustrated under Figure 3.3; the final outcome of an adaptation cycle, being the activation / cancellation decisions, are emphasised with a large dashed arrow.

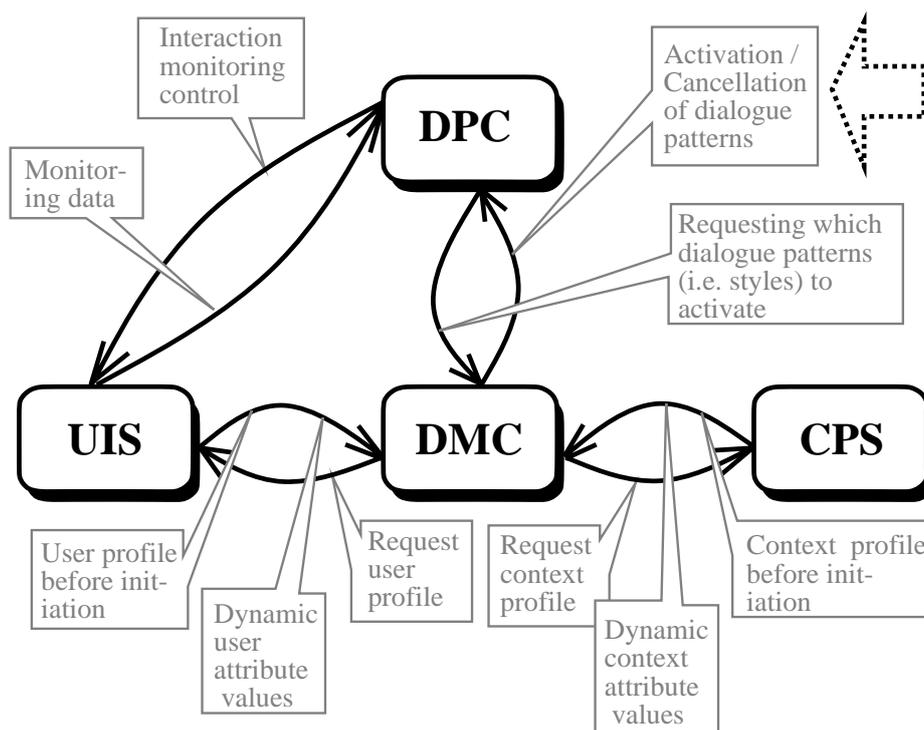


Figure 3.3 - Communication requirements among the components of the unified architecture in order to perform interface adaptation cycles.

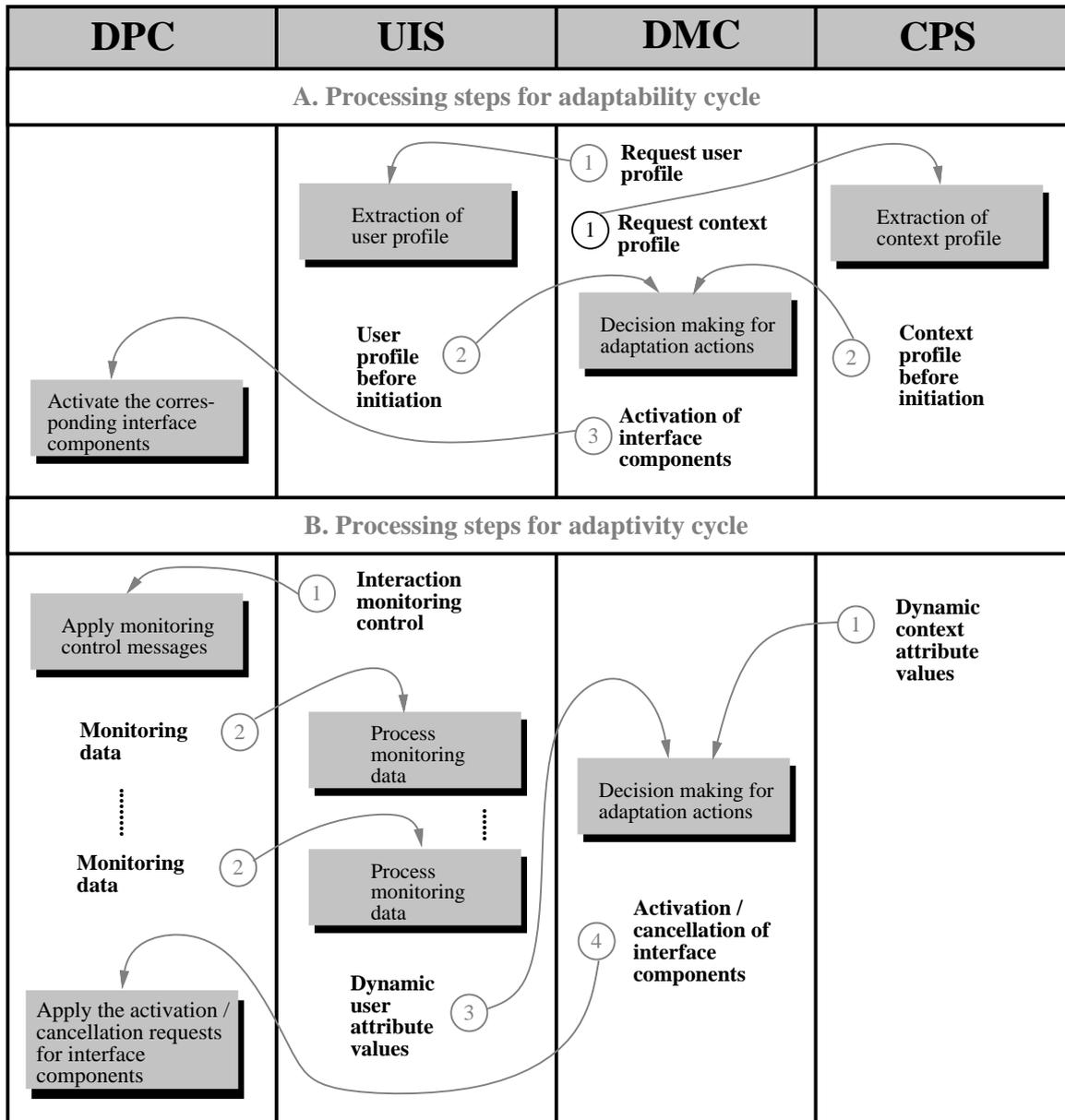


Figure 3.4 - Processing steps, engaging communication among architectural components, to perform the initial adaptability cycle (A), as well as the two types of adaptivity cycles (B); the messages communicated among components are labelled with their logical ordering, while internal processing points are indicated with shaded rectangles.

In Figure 3.4, the processing steps are outlined for performing both the initial adaptability cycle (to be executed only once), as well as the two types of adaptivity cycles (i.e. one starting from “dynamic context attribute values”, and another starting from “interaction monitoring control”). Local actions indicated within components (in each of the four columns) are either outgoing messages, shown via bold typeface, or necessary internal processing, illustrated via shaded rectangles. For each component, the actions “flow” logically on the vertical direction, within its corresponding column; however, Figure 3.4 is not to be also used to compare the logical ordering of actions across components (i.e. the fact that an action in one component, is vertically drawn lower than a particular action in another component, does not imply that the former action precedes in time the latter).

3.3.2 Detailed Communication Semantics

We will present the communication protocol among the various components, in a form mainly emphasising the regulations governing the exchange of information among various communicating parties, as opposed to a strict message-packet syntax description. Hence, our primary focus will be on the semantics of communication, regarding: (a) the type of information communicated; (b) the content it conveys; and (c) the usefulness of the communicated information at the recipient component side.

In the unified software architecture (outlined in Figure 3.1) there are four distinct bi-directional communication channels, each engaging a pair of communicating components. For instance, one such channel concerns the communication between the User Information Server (UIS) and the Decision Making Component (DMC); each channel defines two protocol classes, e.g. $UIS \rightarrow DMC$ (i.e. type of messages sent from UIS to DMC) and $DMC \rightarrow UIS$ (i.e. type of messages sent from DMC to UIS). The description of the four communication channels for the unified interface architecture follows, defining four pairs of protocol categories.

3.3.2.1 User Information Server versus Decision Making Component

In this communication channel, there are two communication rounds: (a) prior to initiation of interaction, the Decision Making Component requests the user profile from the User Information Server, which replies directly with the corresponding profile (as a sequence of attribute values); and (b) after initiation of interaction, each time the User Information Server detects some dynamic user attribute values (on the basis of interaction monitoring), it will communicate those values immediately to the Decision making Module. In Figure 3.5, the classes of messages communicated between the User Information Server and the Decision Making Component are defined, with simple examples.

UIS→DMC	
Message class	Exporting an end-user profile.
Content structure	Sequence of <Attribute, Value> pairs.
When communicated	When requested by the Decision Making Component.
Example	{ <domain knowledge, limited>, <visual ability, sighted>, <age, 35>, <motor ability, fine>, ... }
<hr/>	
Message class	Dynamic detection of user parameters.
Content structure	<i>Parameter, Content.</i>
When communicated	Each time an inference is made by the User Information Server.
Example	{ user confused, Link Selection Task }
DMC→UIS	
Message class	Requesting user profile.
Content structure	This message contains no data (only the message class header).
When communicated	Before decision making is initiated in the Decision Making Component.
Example	{ }
<hr/>	
Message class	Requesting explicitly the value of a user attribute.
Content structure	<i>Attribute.</i>
When communicated	As needed by the Decision Making Component.
Examples	{ age }, { domain knowledge }, { Web knowledge }

Figure 3.5 - Communicated messages between the User Information Server (UIS) and the Decision Making Component (DMC).

3.3.2.2 User Information Server versus Dialogue Patterns Component

The communication among these two components aims to enable the User Information Server (UIS) to collect interaction monitoring information, as well as to control the type of monitoring to be performed. The User Information Server may request monitoring at three different levels: (a) *task* (i.e. when *initiated* or *completed*); (b) *method* for interaction objects (i.e. which logical object action has been accomplished - like “pressing” a “button” object); and (c) *input event* (i.e. specific device event - like moving the mouse or pressing a key).

UIS→DPC	
Message class	Controlling the level of monitoring.
Content structure	<i>Status, Level.</i> <i>Status = on or off.</i> <i>Level = TaskLevel or EventLevel or MethodLevel.</i> <i>EventLevel = event, ObjectName, EventCategory.</i> <i>TaskLevel = task, TaskName.</i> <i>MethodLevel = method, ObjectName, MethodCategory.</i>
When communicated	As needed by the User Information Sserver inference mechanisms.
Examples	<pre>{ on, task, Link Selection } { off, event, HTMLPageWindow, KeyPress } { on, method, BWDPageButton, Pressed } { on, method, HTMLPageScrollbar, Scrolled }</pre>
DPC→UIS	
Message class	Monitoring data.
Content structure	<i>TaskBased or EventBased or MethodBased.</i> <i>TaskBased = task, TaskName, TaskAction.</i> <i>TaskAction = initiated or completed.</i> <i>EventBased = event, ObjectName, EventCategory, EventData.</i> <i>EventData = Sequence of <Parameter, Value> pairs.</i> <i>MethodBased = method, ObjectName, MethodCategory.</i>
When communicated	When the corresponding user actions are performed.
Examples	<pre>{ task, Link Selection, initiated } { task, Link Selection, completed } { event, PageLoadingControlToolbar, KeyPress, <key,"a"> } { event, BWDPageButton, MouseButtonPress, <Button, 2> } { method, StopLoadingButton, Pressed } { method, ReloadButton, Pressed }</pre>

Figure 3.6 - Communicated messages between the User Information Server (UIS) and the Dialogue Patterns Component (DPC).

In response to monitoring control messages, the Dialogue Patterns Component will have to: (a) activate / disable the appropriate interaction monitoring software modules; and (b) continuously export monitoring data, according to the monitoring

levels requested, back to the User Information Server (initially, no monitoring modules are activated by the Dialogue Patterns Component). In Figure 3.6, the classes of messages communicated between the User Information Server and the Dialogue Patterns Component are defined, with simple examples.

3.3.2.3 Decision Making Component versus Dialogue Patterns Component

The dialogue patterns are organised within the Dialogue Patterns Component in the following manner: Suppose that alternative dialogue patterns are designed for a particular user sub-task, addressing distinct values of the decision parameters (i.e. user- and context- parameters). Then, as part of the software implementation, each such alternative dialogue pattern is associated to its respective sub-task, while it is given an arbitrary indicative name, unique among the dialogue patterns of its corresponding sub-task; in the unified design method, designers are directed to provide names indicative of the style of interaction supported by each dialogue pattern.

For this reason, in the unified design method, the various alternative dialogue patterns of a particular sub-task are called simply *styles*; we will follow, from now on, this convention; thus, the term style, will refer to the implemented dialogue patterns which realise alternative designs for user sub-tasks.

At start-up, before initiation of interaction, the Dialogue Patterns Component will request (for each user-task) the names of the *styles* (i.e. implemented dialogue patterns) which have to be activated, so as to realise the adaptability behaviour; the Decision making Component will trigger the adaptability cycle (as illustrated in Figure 3.4), and will respond accordingly (at the end of the adaptation cycle, see Figure 3.4, part A).

After initiation of interaction, the Decision Making Component may “take the initiative” to communicate dynamic style activation / cancellation messages to the

Dialogue Patterns Component; such a communication always occurs at the end of each adaptivity cycle (see Figure 3.4, part B). In Figure 3.7, the classes of messages communicated between the Decision Making Component and the Dialogue Patterns Component are defined, with simple examples.

DPC→DMC	
Message class	Requesting active styles for a particular sub-task.
Content structure	<i>TaskName.</i>
When communicated	Upon start-up, to initiate the necessary dialogue patterns.
Examples	{ Link Selection }, { Page Loading Control } { Page Display Control }
DMC→DPC	
Message class	Posting decisions regarding style activation / cancellation.
Content structure	<i>StyleDecision=DecisionType, StyleSignature.</i> <i>DecisionType= activation or cancellation.</i> <i>StyleSignature = TaskName, StyleName.</i>
When communicated	<ul style="list-style-type: none"> • At the end of the adaptability cycle, prior to initiation of interaction. • At the end of each adaptivity cycle, after initiation of interaction..
Examples	{ activation, Link Selection, Direct } { activation, Stop Loading, With-Confirmation } { cancellation, Link Selection, Direct } { activation, Link Selection, With-Confirmation }

Figure 3.7 - Communicated messages between the Decision Making Component (DMC), and the Dialogue Patterns Component (DPC).

3.3.2.4 Decision Making Component versus Context Parameters Server

The communication between these two components is very simple: the Decision Making Component will request the various context parameters values (i.e. usage-context profile), while the Context Parameters Server will respond accordingly. During interaction, dynamic updates on certain context property values are to be also communicated to the Decision Making Component for further processing (i.e. possibly new inferences will be made).

As previously mentioned, the Context Parameters Server aims to support two levels of functionality: (i) encompasses information regarding the available I / O facilities (i.e. a type of system “registry”) on end-user machine; and (ii) monitors particular environment parameters, such as environment noise or user presence in front of the

terminal (e.g. via infrared sensors). The first category of information is necessary for deciding, at the Decision Making Component side, those interaction techniques which: on one hand conform to user attribute values, and on the other hand can be fully supported via the peripheral equipment at the end-user terminal. The second category is necessary for many purposes.

DMC→CPS	
Message class	Requesting context parameter values.
Content structure	This message contains no data (only the message header).
When communicated	Prior to initiation of interaction, beginning of adaptability cycle.
Examples	{ }
CPS→DMC	
Message class	Responding with a list of usage-context parameter values.
Content structure	Sequence of <Attribute, Value> pairs.
When communicated	When requested by the Decision Making Component.
Examples	{ <Screen Resolution, <640, 480>>, <Mouse Available, YES>, <Terminal Position, <120, "cm">>, <Software volume ctrl, YES>, <Speech Synthesis, YES> }
Message class	Dynamic usage-context attribute values.
Content structure	Attribute, Value.
When communicated	Each time a usage-context attribute value is dynamically modified.
Examples	{ Environment noise, 65 dB } { User in front of terminal, NO } { User in front of terminal, YES }

Figure 3.8 - Communicated messages between the Decision Making Component (DMC), and the Context Parameters Server (CPS).

For instance, it may be utilised in order to provide interaction not conflicting with the particular environment state; e.g. may avoid audio feedback, if high environment noise is detected. Another scenario is for supporting the inference process of making dynamic assumptions about the user; e.g. if notification is received that the user moves away from the terminal, then that particular interaction session is terminated; otherwise, idleness for a long period of time, while the user is still in front of the terminal, could be interpreted as a potential “confusion”, “loss of orientation” or “inability to perform the task”. In Figure 3.8, the classes of messages communicated between the Decision Making Component and the Context Parameters Server are defined, with simple examples.

3.4 COMPONENT IMPLEMENTATION ISSUES

In this Section, we will provide the type of development strategies / tools that we consider as more appropriate for each architectural component; such guidelines have been consolidated from real-life experience with medium- to large- scale unified interface development projects, such as a hypermedia information system [Petrie, et al, 1996] supporting dual interaction [Savidis et al, 1995a], an augmentative communication system for language-cognitive- and speech-motor- impaired people [Kouroupetroglou et al, 1996], and an adaptable and adaptive Web browser for HTML 3.2 [Stephanidis, et al, 1997b]. We will briefly analyse the component implementation issues by means of which tool categories are considered as more appropriate “to do the job”, rather than addressing specific algorithmic aspects or software engineering methodologies.

Also, for some of the components, more elaborated architectural patterns will be proposed. These can be treated as generic design patterns [Gamma et al, 1995], and provide an implementation structure for building those components, so that the desirable functional behaviour can be effectively accomplished.

3.4.1 User Information Server - *Implementation Issues*

A typical knowledge representation approach may be employed for representing user models and profiles, as well as for drawing assumptions about particular user attributes at run-time (i.e. during interaction). An appropriate local (to the UIS component) user representation method may be defined and adopted, both for storage of user information as well as for manipulation via the various inference engines; however, the user representation should be always converted to the general form of attribute / pairs when communicated to the “external world” (i.e. the rest of the

architectural components). In some cases, the User Information Server may need to merely play the role of a user profile repository. In such situations, a minimalistic implementation approach can be taken, where: (a) a data-base of user profiles is kept; (b) a small implementation shell is added to access such a data-base, for processing communication requests. If more than a repository is needed, the User Information Server should be at least capable of receiving and processing (i.e. reasoning) interaction monitoring information; the best marriage for this purpose is the employment of a logic programming language which includes inter-process communication functionality (normally through language add-ons).

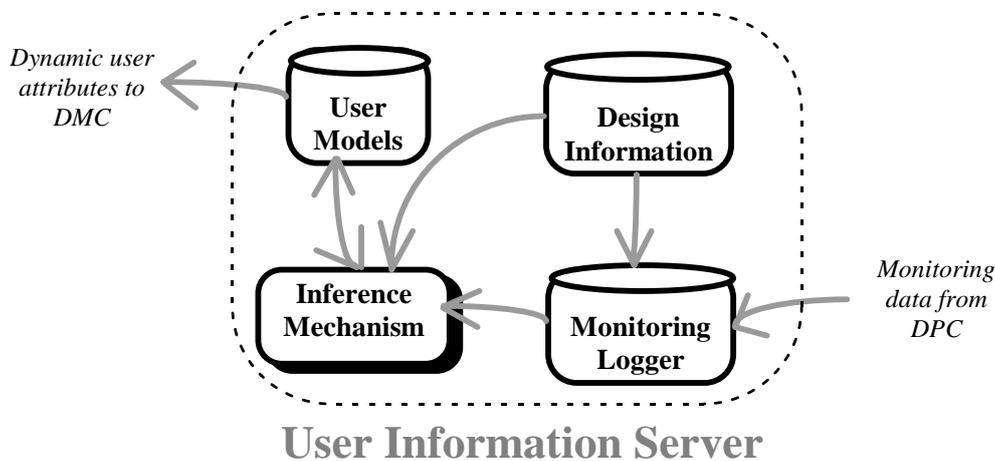


Figure 3.9 - The architectural pattern for the User Information Server.

From the analysis of various aspects of existing adaptive systems [Dieterich et al, 1993], with respect to the employment and utilisation of user modelling methods, an appropriate architectural design pattern for the User Information Server has been designed. This architectural pattern is illustrated under Figure 3.9 and consists of four logical components: (i) The *Monitoring Logger*, which files interaction monitoring events (i.e. stores events from the interaction history); the modules performing such monitoring reside in the interactive software. (ii) The *User Model*, which is needed to drive an adaptation-oriented decision making process; the individual user models (or profiles) may be locally kept within this component. (iii) The *Design Information*, providing design-related knowledge, which is necessary for both associating

interaction monitoring data with design context, as well as for supporting dynamic user attribute detection. (iv) The *Inference Mechanism*, which encompasses all the necessary knowledge to derive, during interaction, user attribute values (e.g. preferences, disorientation, expertise).

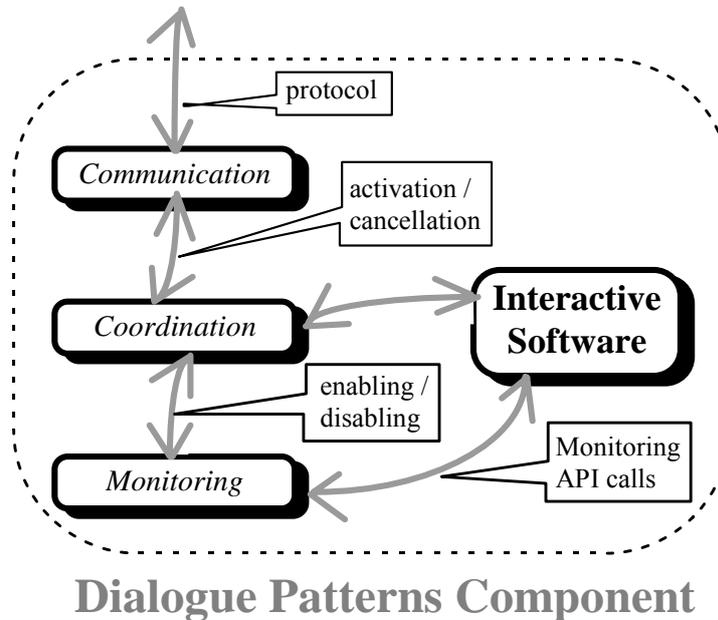


Figure 3.10 - Expanding existing interactive software to constitute the Dialogue Patterns Component in unified system architecture.

3.4.2 Dialogue Patterns Component - *Implementation Issues*

This component should encompass all the various dialogue patterns in an implementation form. In this sense, typical development methods for building interactive software may be freely employed; the unified architecture poses no restrictions on the type of interface tool being utilised. Additionally, some special purpose functionality needs to be introduced on top of interactive software implementing the designed dialogue patterns, realising the “packaging” (of interactive software) as the Dialogue Patterns Component of a unified system architecture. Such extra required functionality is split into three categories (see

Figure 3.10): (i) *communication* with the rest of architectural components (i.e. receiving / sending messages, following the inter-component communication semantics); (ii) *monitoring*, encompassing interaction monitoring code, which is to be attached to various interface components; and (iii) *coordination*, capable of applying activation / cancellation decisions on interface components (as received from DMC), and enabling / disabling interaction monitoring components (as received from UIS). This is the only type of functionality required to make existing interactive software available in the context of a unified architecture. It should be noted that the employed interactive software may internally encompass architectural links in the context of a particular software framework, such as an Arch-based interactive structure, as shown within Figure 3.1. All such possible architectural connections are not damaged by the software expansion approach of Figure 3.10, since the original interactive software is by no means affected (i.e. *orthogonality* principle). In practice, there may be some lower-level issues which need to be attacked:

- *Attaching interaction monitoring software as independent software modules may present implementation barriers, requiring monitoring code to be “injected” within original software implementation.* This implementation barrier has been recently overcome, since all widely development instruments are built on top of platforms (e.g. Java AWT / Beans, MFC or ActiveX) which explicitly provide monitoring Application Programming Interfaces (e.g. Active Accessibility™, by Microsoft, and Java Accessibility™, by JavaSoft).
- *When expanding interactive software to encapsulate adapted behaviours, there are some designed dialogue patterns not made available in an implementation form.* This is naturally expected, if new user- or usage-context- parameter values (not originally foreseen) are taken into consideration, thus resulting in the construction of new design artefacts. In such cases, a dedicated implementation process should be initiated, to make all extra dialogue patterns available in an appropriate implementation form.

- *Various interface components are built via non-programming oriented tools, not explicitly visualising to interface developers an Application Programming Interface, through which run-time activation / cancellation could be programmed.*
The answer to this issue is that all such widely used tools (e.g. Visual Basic, Java Beans interactive construction tools, ActiveX components builders) generate explicitly code, through which programmatic access on interface components is also allowed; in many cases, developers preferring WYSIWYG (What You See Is What You Get) construction methods are not aware of those techniques, which, however, are well documented and are usually accompanied with rich functionality.

3.4.3 Decision making Component - *Impementation Issues*

This component encompasses design logic representation into a form which, for each sub-task, appropriately associates user- and context- attribute values with the various implemented alternative (if any) styles (i.e. dialogue patterns). Typical logic / knowledge programming languages are most appropriate for implementing this kind of functionality. For applications with simple decision making logic, common programming languages may also suffice for hard-coding the decision process; however, this is only suggested in the case of simple “if-else” logic. In case of more complex design relationships and design-based reasoning, a knowledge representation framework should be employed.

3.4.4 Context Parameters Server - *Implementation Issues*

This component is expected to be at the level of system software, though simple in complexity, accessing external peripheral equipment (in case of monitoring) and providing information on available I / O devices (i.e. some kind of a simple “registry” for installed peripheral equipment). This component is also responsible for usage-context monitoring. The idea of monitoring the environment in which the user is engaged, has been practically applied in the context of smart-home technology, in order to mainly identify alarm situations, by processing the input from various types of special-purpose sensors. Due to the lower-level functional requirements of this components, a typical third generation programming language, like C, is considered as more suitable.

3.5 DISCUSSION

The unified development paradigm has been designed so as to reflect the functional properties of automatically adapted behaviours, by supporting a clear separation of development roles in a fully distributed component-based architecture. The notions of dialogue pattern, decision making, and user- and context- attributes are directly mapped to computable run-time constructs. The unified development paradigm, apart from driving the construction of automatically adapted dialogues, exhibits some additional distinctive properties which will be briefly discussed: incremental development, support for re-usability, appropriateness for component-based technologies, and openness with respect to emerging Web-based application development paradigms.

3.5.1 Incremental Development

The notion of incremental development is related to the need of modifying or extending an existing interactive application. In case that new design parameters need to be addressed, due to consideration of additional user- or context- parameters and / or the introduction of new user tasks, appropriate design activities need to be executed, inherently requiring updates on the implementation code. In the unified development paradigm, the support for incremental development is fundamental.

The consideration of extra user- or context- parameters can be carried out in a straightforward manner: the new decision parameter instances will lead to potentially new dialogue patterns for various sub-tasks. If the user tasks are also extended (i.e. adding extra interactive functionality), then new categories of dialogue artefacts will emerge, associated to the newly introduced sub-tasks. Additional dialogue patterns will have to be implemented and installed within the Dialogue Patterns Component, while the new associated rules for decision making will need to be embedded within the Decision Making Component. Such support for incremental development is critical particularly in the context of user-adapted interaction, where it is expected that the need for addressing new user parameters will emerge, after some initial development cycles have been completed.

Interface design pattern	Interface implementation pattern
<ul style="list-style-type: none"> • User tasks • User attributes • Context attributes 	<ul style="list-style-type: none"> • Implementation module • Functional properties • APIs
<p><i>No explicit semantic association exists between the parameters driving identification of potential reuse</i></p>	

Figure 3.11 - Semantic gap between design and implementation patterns towards reuse.

3.5.2 Reusable Dialogue Patterns

Even though an interactive software product realises a well documented interface design, it is unusual to identify syntactic concepts and design entities, including their associated relationships, with their respective names from their design context, when inspecting the implementation code. Only programming object hierarchies, related to the lexical design layer, have a more close association to the design “world”. The reason is that the transition from the design phase to the implementation phase is carried as a continuous implementation-oriented interpretation of design artefacts, thus leading to loss of design information, as we get closer to the implementation “world”. Such loss of design information occurs in the following two levels of design interpretation: (i) designers have to translate the design outcome into a form that programmers will understand; and (ii) programmers interpret the design documentation reported by designers with an implementation-oriented approach.

This situation has resulted in two different perspectives for re-usability in the context of interactive systems: (a) design re-use; and (b) implementation re-use. The criteria defining re-usable patterns in these two “worlds” are, unfortunately, radically different. As a result, there is a large gap between design patterns and implementation patterns, when dealing with the notion of re-usability (see Figure 3.11). In the unified implementation method, the association of design patterns to implementation components is explicit: the Decision Making Component manipulates design knowledge regarding dialogue patterns, while the various activation / cancellation decisions concern dialogue patterns as defined in the design representation world; on the implementation side, the Dialogue Patterns Component “knows” how to map design patterns to implemented interactive components, thus activating or cancelling the correct run-time modules. Hence, the unified architecture supports the direct association of design concepts and artefacts with their respective implementation-oriented constructs. We characterise this capability as *design-based implementation re-use*, and it supports design-oriented software engineering at the interface implementation domain.

The main design components which may directly “constitute” implementation patterns, are the alternative dialogue patterns (i.e. styles) for each sub-task, which are introduced due to diverse user- and context- parameter values (i.e. decision parameter

instances). Such implementation patterns, due to the fact that they can be directly indexed on the basis of their respective sub-task decision parameters instances (i.e. user- and usage-context- attribute values addressed), can be deposited in the context of an implementation pattern repository and subsequently located and retrieved through their indexing parameter values.

3.5.3 Employing Componentware Technologies

The component-based development paradigm is based on the deployment of implemented software from multiple sources, directly as it is (i.e. in binary format, not only as source code), by supporting a software assembly process resembling that of hardware manufacturing. The underlying driving principles are that of *reusability* and *reliability* (i.e. re-using extensively tested components). Currently, there are software technologies supporting component construction and deployment, such as CORBA (by OMG), OpenDOC (by Apple), DCOM / ActiveX (by Microsoft), and Java Beans (by JavaSoft).

The unified development approach promotes the componentware paradigm in two levels: (i) at the *architectural level*, visualising explicit components which may be re-used across applications (i.e. the Decision Making Component, the User Information Server and the Context Parameters Server); and (ii) at the *dialogue patterns level*, since it provides an implementation model in which dialogue patterns can be directly mapped to distinct software components that can be located on the basis of directly computable design parameters (i.e. sub-task, user-, and usage-context- attribute values).

Currently, in the domain of work-flow systems for enterprise network computing, there is an increasing trend towards the production of components for common business tasks [Short, 1997]. Such a task-oriented component production, reflecting primarily the design targets of packaged components, is in full compatibility with the unified development paradigm, which additionally, introduces extra parameters, further specialising the design scope of component dialogue patterns.

3.5.4 Remote Downloading and Application Assembling - *the Web Future*

The component-based and distributed computing paradigms have been “pushed” forward considerably by the introduction and broad use of the World Wide Web. The Web has provided a new implementation platform for client / server computing, and it is expected to largely promote componentware in the future; but software must be better modularised to support this evolution [Short, 1997]. It must be delivered as well-behaved components, whose purpose (i.e. what goal they serve) can be easily identified, and which can be easily connected to other components. *Web browsers will be enhanced to become application assemblers* [Short, 1997]. Following this trend, interface development seems to change dramatically, continuing the paradigm shift which has already been performed by the introduction of the Web. The old development paradigms need to be revisited, while interactive software has to be broken down into meaningful pieces, serving specific design objectives and work-tasks.

The interactive software development process may demonstrate numerous ways of putting pieces together, according to the particular instances of adaptation decision parameters. From the early age of hypertext-oriented “executable context”, Web documents have recently evolved to more sophisticated structures, conveying content in various forms (see Figure 3.12). Such a progress has effectively resulted in Web documents with increased interactive features. In this context, taking into consideration the tremendous growth of Web users within the last years, the

introduction of software methods to enhance the quality of produced Web-based interactive software becomes more prominent than ever before. The application of the unified development paradigm on the Web infrastructure should only concern the interactive, as well as semi-interactive elements (see Figure 3.12).

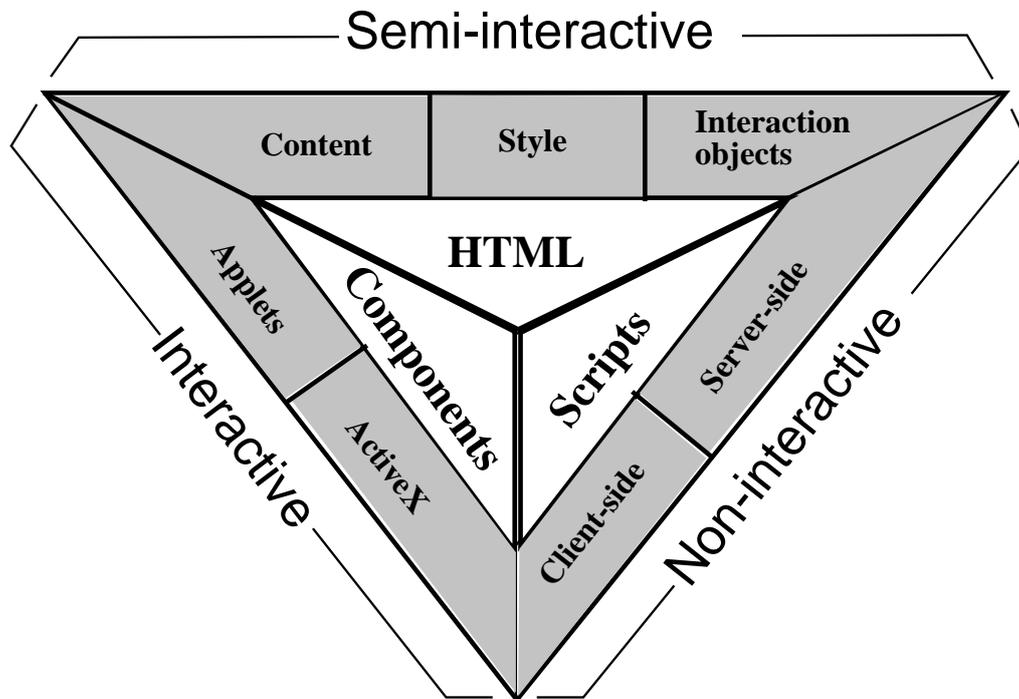


Figure 3.12 - The three main content categories of Web documents.

In particular, one promising field is the fusion of intranet-based software architectures for remote application assembling, with our unified interface development paradigm, in the context of enterprise network computing applications. The intranet paradigm is very powerful, and it becomes practically feasible even for the broader internet with the employment of advanced techniques to enhance downloading time, or reduce the load on application servers (e.g. replication of application servers, multiple application servers connecting to a central application server for software updates). In this paradigm, users are faced with software applications remotely broadcast, assembled on the fly, and always kept up-to-date. The software interface of interactive products is also made-up of components pulled from an application server.

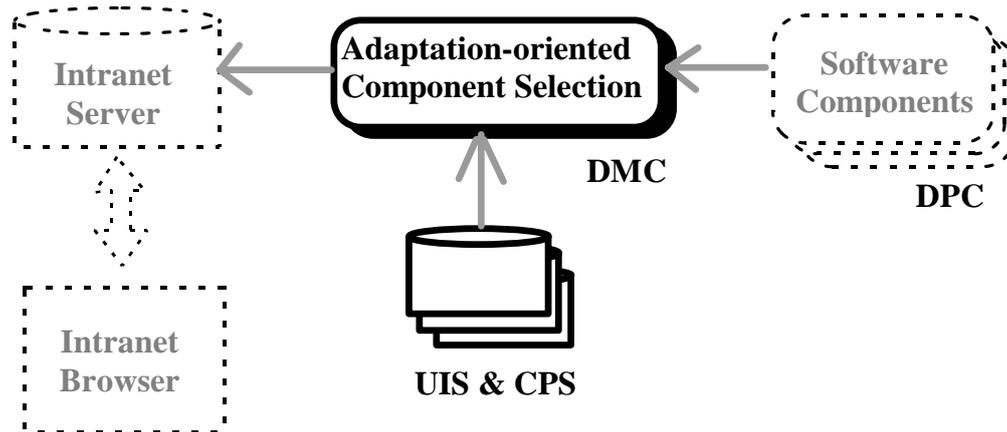


Figure 3.13 - Adaptation-oriented component selection at the Intranet Server, leading to a blending of the unified architecture with intranet-based distributed applications; the mapping to unified architectural components is also indicated.

Currently, it is the responsibility of application developers to choose the necessary interface components, being mostly appropriate for the particular work-tasks, and manually organise them by means of integrated software products, maintained at application servers; this is the typical scenario of a component-based development process. But the evident “explosion” in component industry has already come-up with a considerable number of components for various work-tasks.

Such alternative components vary both by means of underlying functionality, as well as of their interaction means. Such different dialogue design approaches for available software components, when referring to similar work-tasks, have emerged due to consideration of differing user attributes, preferences, organisational roles in an enterprise, or even contexts-of-use (e.g. machines, environments).

In order to support high quality of interaction, the need for software mechanisms supporting an automatic component selection process is evident, combining knowledge on end-users and contexts-of-use, with domain-oriented design logic and design criteria. Such software mechanisms may inter-operate with conventional intranet application servers, working prior to the local assembly process at the browser-side, by collecting and supplying to application servers those specific

components suitable for accomplishing high-quality adapted interaction. This technical approach is illustrated in Figure 3.13, indicating the additional component selection layer added at the server side of intranet establishments.

Chapter IV

FUNCTIONAL REQUIREMENTS FOR UNIFIED DEVELOPMENT

4.1 INTRODUCTION

The implementation of unified interfaces is a very demanding task. It should enable the manipulation of diverse categories of interaction elements, support the construction of dialogue components employing diverse interaction means and even comply to different interaction metaphors, while also promoting openness, extensibility and portability. We will focus on the issue of implementing the various dialogue patterns from the outcome of a unified design process, revealing the most important implementation mechanisms which are required, when developing such dialogue artefacts for diverse users and usage-contexts.

4.1.1 Key Technical Requirements

4.1.1.1 New Interaction Metaphors

The introduction of new interaction metaphors is prominent if existing metaphors do not suffice for supporting high-quality interaction, with respect to particular user groups and / or application domains. For instance, the development of non-visual interaction on the basis of the desk-top metaphor can be criticized due to the fundamental visual orientation of the original desk-top metaphor. Also, the employment of conventional desk-top interaction facilities (i.e window-based interaction) for educational applications targeted to children, has been inherently substituted by new highly interactive applications realizing and enhancing various real-world metaphors (e.g. play-room, living books). Currently, the design and realization of various new specialized interaction metaphors is not followed by corresponding re-usable software libraries and toolkits, but it is rather hard-coded within particular large software applications. Even though various educational applications provide virtual worlds familiar to children as an interaction context, the lack of the required "world" construction kits for such specialized domains and metaphors is lacking, thus necessitating that developers undertake this considerably large development overhead. Existing multimedia application construction libraries are

too low-level, requiring that developers build all metaphoric interaction features (it is analogous to the case of starting with a basic graphics package for building window-based interactive applications). It is evident that the availability of new metaphors, as well as the provision of the necessary toolkits, is crucial for promoting the commercial development of new applications, targeted to the broad population at large.

This remark reveals the need of developing new metaphors, inherently requiring all the necessary implementation support within interface tools. Efforts devoted to the development of new metaphors contribute towards broadening the interaction technologies domain (i.e. resource dimension) of unified interfaces. We will provide a metaphor development methodology, as part of the unified interface engineering paradigm; additionally, a specific design and implementation case for non-visual interaction, carried out in the context of this thesis, will be presented.

4.1.1.2 Effective Manipulation of Interaction Objects

Interface development tools play a key role in the User Interface development life cycle, and their functional capabilities may affect considerably the eventual quality of interactive software products. The vast majority of commercially available tools is targeted towards the implementation phase, while some of them also provide support for design-oriented development activities. In all cases, interface tools provide development facilities in which the notion of interaction elements, as the basic interaction building blocks, is directly mapped to explicit constructs. The most typical categories of such interaction elements are: (i) interaction objects / interaction techniques / object hierarchies; (ii) input events; (iii) graphic primitives; and (iv) callbacks / methods / notifications.

Usually, instances of such interaction element categories are provided in an implementation form by software libraries called *toolkits* (e.g. OSF / Motif, WINDOWS Object Library, InterViews, Xaw / Athena widget set). Interaction objects (e.g. windows, buttons, check-boxes) are the most important interaction element category, since the largest part of existing interface development toolkits is

devoted to providing rich sets of interaction objects, accompanied with all the necessary functionality. Interaction objects are communicated at both the design as well as the implementation domain; in other words, interface designers are well aware of the various interaction objects classes and their respective “look & feel”, while programmers also share this type of knowledge. Naturally, designers have more detailed knowledge regarding their appropriateness for particular user tasks, while programmers have primarily implementation-oriented knowledge. In any case, a “button”, or a “window” has the same meaning for both designers and programmers, when it comes to the physical entity being represented. This is a distinctive property, which is, unfortunately, hardly met in other types of design entities. For instance, user tasks do not directly map to available implementation constructs, hence, programmers would need an explicit mapping of a task model to an appropriate structure being closer to their implementation world. This mapping, unless automated by a tool, introduces an extra overhead; furthermore, the resources spent for such an activity are not meant to increase the eventual interface quality.

In recent development paradigms, such as the Web infrastructure for distributed interactive hypermedia documents, the importance of interaction objects has been particularly demonstrated. In HTML 3.0 (and included in all subsequent versions), “Form” elements support a comprehensive collection of interaction objects; also, in the JAVA language, the AWT library (Abstract Window Toolkit) provides a rich set of interaction objects supporting *retargetability* (i.e. mapping to multiple graphical platforms). In conclusion, interaction objects, as re-usable interaction building blocks, are currently considered as a widely accepted and applied paradigm, for the design and implementation phases.

The functional needs for manipulating interaction objects have been studied in the context of real-life application development ([Petrie et al, 1996], [Savidis et al, 1995a], [Savidis et al., 1995c], [Stephanidis et al., 1997b]), targeted towards diverse user groups, differing with respect to various parameters such as: physical / mental / sensory abilities, preferences, domain-oriented knowledge, role in organisational context, etc. In this context, various interaction technologies and interaction

metaphors had to be employed, like: windowing graphical environments, auditory / tactile interaction, Rooms-based interaction metaphors, etc. We summarise below our key remarks from this study, concerning the manipulation of interaction objects in interface tools.

- Need another additional toolkit (i.e. *integration* / *importing* of a particular software library is needed);
- Dialogue for objects, as provided by the toolkit(-s) being used, is not adequate (i.e. *augmentation* with extra interaction techniques is imposed);
- Some necessary interaction objects are not provided from the toolkit(-s) being utilised, and / or new custom-made interaction objects are designed (i.e. *expansion* of the particular toolkit is required);
- Require manipulation of objects at a level “higher” than the typical implementation layer of toolkits, in order to make the dialogue design applicable to multiple user groups and target toolkits (i.e. need *abstraction*, applied on interaction objects).

From the above remarks, the need of the following four mechanisms for manipulating interaction objects emerges: (i) integration; (ii) augmentation; (iii) expansion; and (iv) abstraction. Those four mechanisms are fundamental for handling interaction objects with interface tools aiming to support unified interface development.

4.1.1.3 Manipulation of Alternative Styles and Support for Architectural Openness

Interaction elements fall in the lexical level of interaction. In unified interface development, the combination of such constructs leads to the implementation of alternative sub-dialogues (i.e. styles). In order to support automatically adapted

behaviour, interface tools should supply the necessary mechanisms enabling developers to build all the run-time control and coordination features required, including the “injection” of interaction monitoring software. Hence, style implementation and manipulation should be effectively supported. In this context, some more detailed functional requirements emerge, due to the need of manipulating via the same implementation mechanisms diverse interaction artefacts (e.g. visual windowing-, 3D auditory-, tactile-, and switch-based- dialogues), which have to be coordinated, as well as linked together with the same semantic non-interactive software layer.

Finally, architectural openness is required, since the vertical growth of software systems to support adapted behaviours, should be effectively accomplished. Additionally, maximal openness may be reached, if direct exchange of implemented dialogue components among interoperating tools is facilitated.

4.1.2 The Seven Key Mechanisms in Unified Development

From the previous analysis, seven fundamental mechanisms have been identified (see also Figure 4.1): (i) metaphor development; (ii) toolkit integration; (iii) toolkit augmentation; (iv) toolkit expansion; (v) toolkit abstraction; (vi) style implementation and manipulation; and (viii) architectural openness. We will incrementally analyse each of the mechanisms as follows:

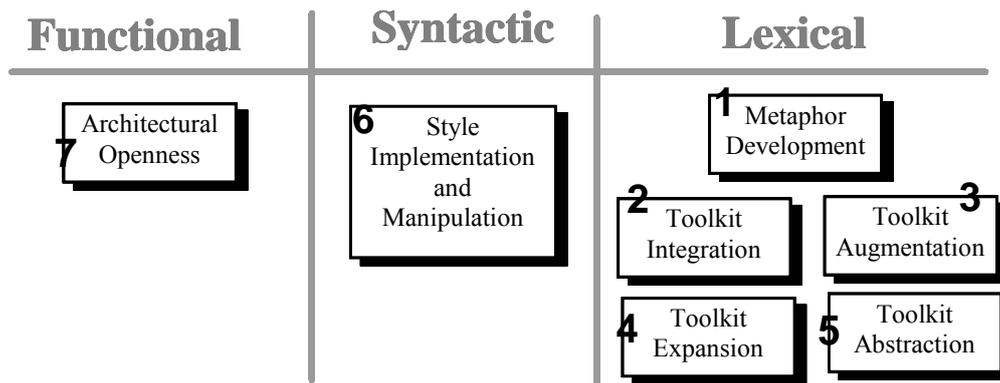


Figure 4.1 - The seven key mechanisms for unified development of interactive software, positioned within the three fundamental interaction layers; the order of discussion of each mechanism is also annotated.

Firstly, we will provide its basic definition and elaborate its importance in the context of unified development. Secondly, we will identify the functional requirements for interface tools, so that the mechanisms can be practically supported. Then, we will study previous work, assessing whether some of the mechanisms have been supported before, and in what degree. Finally, we will discuss in detail how those mechanisms have been explicitly supported within design and implementation efforts carried out as part of this thesis. This last discussion will be mainly focused on: (a) the facilities offered by the I-GET UIMS tool for supporting those mechanisms (i.e. tool support); and (b) specific development efforts and engineering paradigms in which some of the paradigms have been explicitly (i.e. without tool support) embodied.

4.1.3 The I-GET UIMS Tool

The I-GET tool is a User Interface Management System (UIMS, [Myers,1995]), which provides the I-GET language for interface implementation. It has been developed in the context of the ACCESS Project [ACCESS Project, 1996], in order to support unified implementation of interactive software; this concerns only the Dialogue Patterns Component of the unified software architecture, while the I-GET tool provides no functionality for building the User Information Server, the Context

Parameters Server and the Decision Making Component. The I-GET language is a marriage of interface languages with 4th Generation Languages (4GLs). Interface languages are different from mainstream programming languages like C++ and Java in the sense that interaction-specific concepts (e.g. interaction object, attribute, method, event, notification, event handler, object hierarchy) are supported as built-in language constructs. 4GLs exist for various programming paradigms, like procedural, Object Oriented (OO), functional, formal and logic techniques, while they are characterised by the introduction of more declarative algorithmic constructs, than those typically supported in widely spread languages of their respective programming paradigms. The I-GET language is more close to the procedural and OO paradigm, while it introduces various declarative program control constructs such as: preconditions, constraints and monitors.

The choice of providing a language-based development method has been dictated due to the of need of supporting unified interface development. If the manipulation of diverse interaction elements is to be supported, while still being open with respect to integrated interaction toolkits, interactive construction techniques must be excluded. The reason being that, on the one hand a particular interaction technology has to be presupposed in interactive building tools (e.g. OSF/Motif, Windows, X/Athena), while on the other hand, facilities such as abstraction are not supported in interactive construction.

Also, syntax-oriented methods, such as task notations [Hartson et al, 1990] and action grammars [Reisner, 1981], do not suffice for developing unified interfaces, since in unified interaction it should be allowed to potentially have various alternative syntactic designs (possibly being very “close”, but still different), though “fused” together in a single running interface. Other techniques, such as event-based models [Hill, 1986] and state-based methods [Jacob, 1988], have been rejected because of their fundamentally low-level dialogue control approach (more close to programming), as well as due to their lack of support for abstraction and polymorphism, two critical functional features for unified interface development.

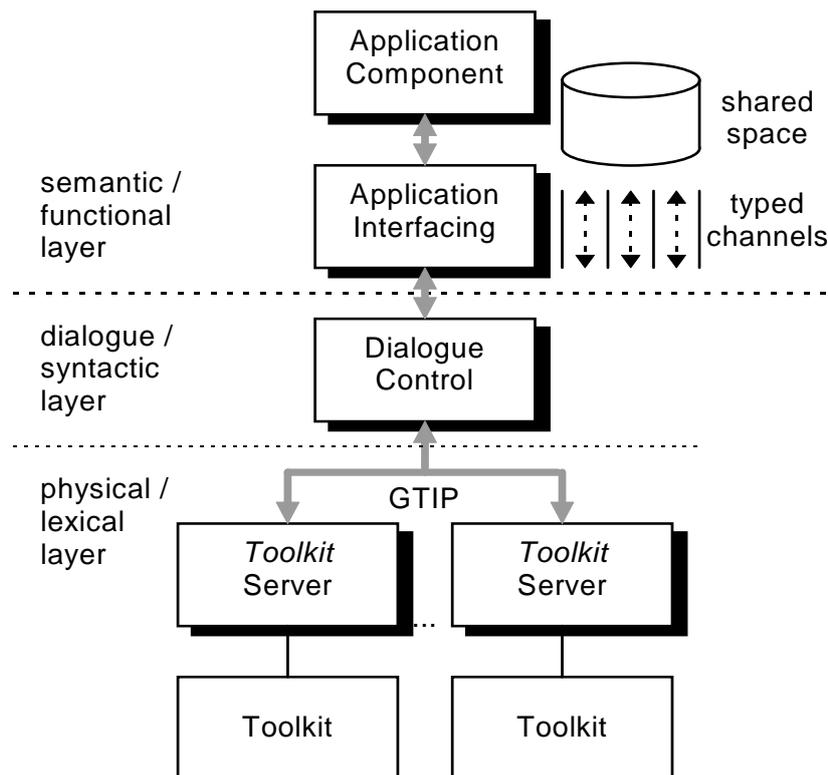


Figure 4.2 - Run-time architecture of interfaces developed through the I-GET UIMS.

4.1.3.1 Run-time Architecture and Development Roles Supported

The run-time architecture of interfaces developed through the I-GET UIMS is illustrated in Figure 4.2. This architecture extends upon the Arch UIMS model, by introducing explicit interfaces among the various components, the most important of which is the Generic Toolkit Interfacing Protocol (GTIP). As it has been discussed, the unified architectural framework can be appropriately combined with those UIMS architectural paradigms; this property allows the direct employment of the I-GET tool for building the Dialogue Patterns Component in a unified system architecture. The detailed explanation of the various architectural components follows.

- A *Toolkit Server* plays the role of an intermediate translator between the Dialogue Control components' "view" of the particular toolkit and the original / native toolkit programming interface. Implementationally, the "real" physical interaction elements will be managed and locally maintained at the toolkit-server side, while

the Dialogue Control components' view consists of lightweight software elements, which gain a physical substance through the toolkit servers' translation "filter".

- The *Toolkit Interface* mechanism of the I-GET tool is not realized via a separate running process. Instead, it is based on a special purpose semantic protocol, called *Generic Toolkit Interfacing Protocol* (GTIP). This protocol has been designed in order to enable physical separation between the programming interface of a toolkit (i.e. object classes, data types and procedures) from the real implementation of physical interaction elements (i.e. rendering, device handling, display algorithms).
- The implementation of the *Dialogue Control* component is realized through the I-GET language, and encompasses all the various interactive modules which are required by the Dialogue Patterns Component of the unified software architecture.
- The *Application Interfacing* component software is provided with the I-GET public release. The role of this module is to coordinate and mediate communication between the Application Component and the Dialogue Control component; in this context, it maintains the space of shared objects and controls "flow" of messages through the message channels.
- *The Application Component* collects together all the various modules supplying non-interactive functionality. The I-GET tool provides a "template" structure requiring implementation of a small set of special-purpose C++ functions, which are the minimal requirements for making an Application Component for the I-GET run-time architecture. An Application Component never communicates directly with the Dialogue Control component, but it may either "access" the shared space or post messages in a message channel. In the same manner, an Application Component is implemented by utilising I-GET library functions for receiving notifications about events occurring in the communication space (which is maintained by the Application Interfacing process).

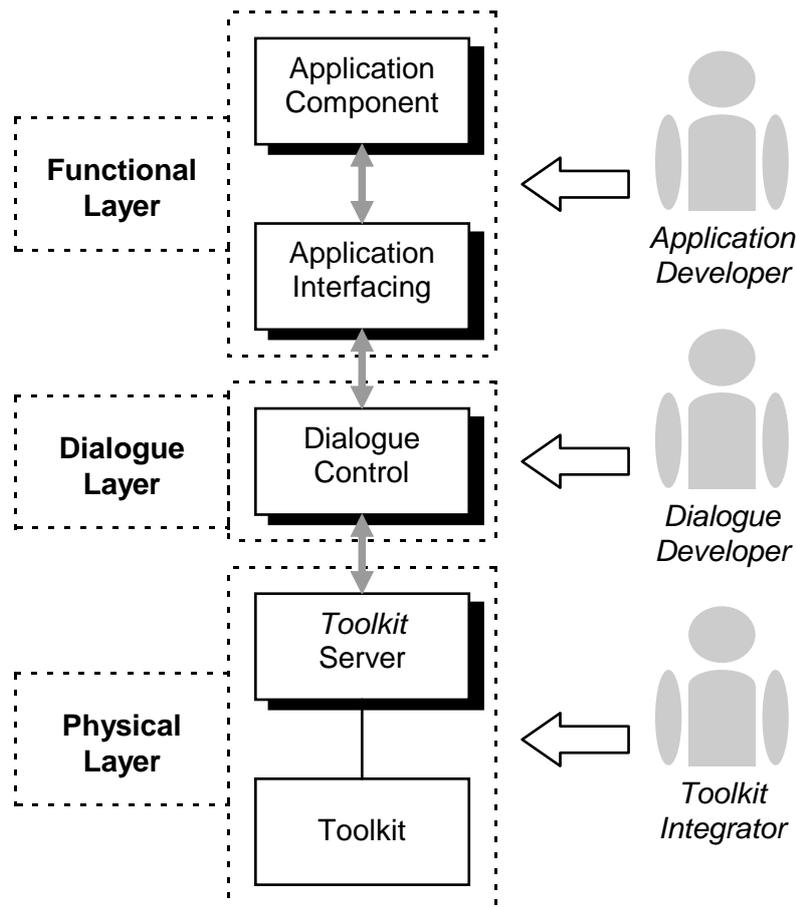


Figure 4.3 - Development roles in the I-GET tool.

The I-GET tool supports different *development tasks* which are logically associated to the following implementation layers (see Figure 4.3): (a) *Functional Layer*, which concerns the development of the *Application Component* and the design & specification of the *Application Interfacing* space; (b) *Dialogue Layer*, which concerns the design & implementation of the *User Interface*; and (c) *Physical Layer*, which concerns the integration of toolkits of interaction elements, a task concerning the toolkit interface specification, as well as the toolkit server development.

4.1.3.2 Layers of Implementation Constructs in the I-GET Language

The I-GET language consists of four key logical layers of constructs illustrated in Figure 4.4: (i) *API layer*, for the specification of the application interfacing space; (ii) *Agent layer*, which concerns the specification of agent classes; agent classes play the role of dialogue control component classes; (iii) *Objects layer*, which is related to the levels of interaction objects supported in the I-GET language; physical objects and virtual objects are both supported; and (iv) *Common layer*, which provides language constructs that can be employed in any of the above three layers: (a) programming kernel; (b) constraints and monitors; (c) hooks and bridges; and (d) prototype / implementation facility, which support separate compilation.

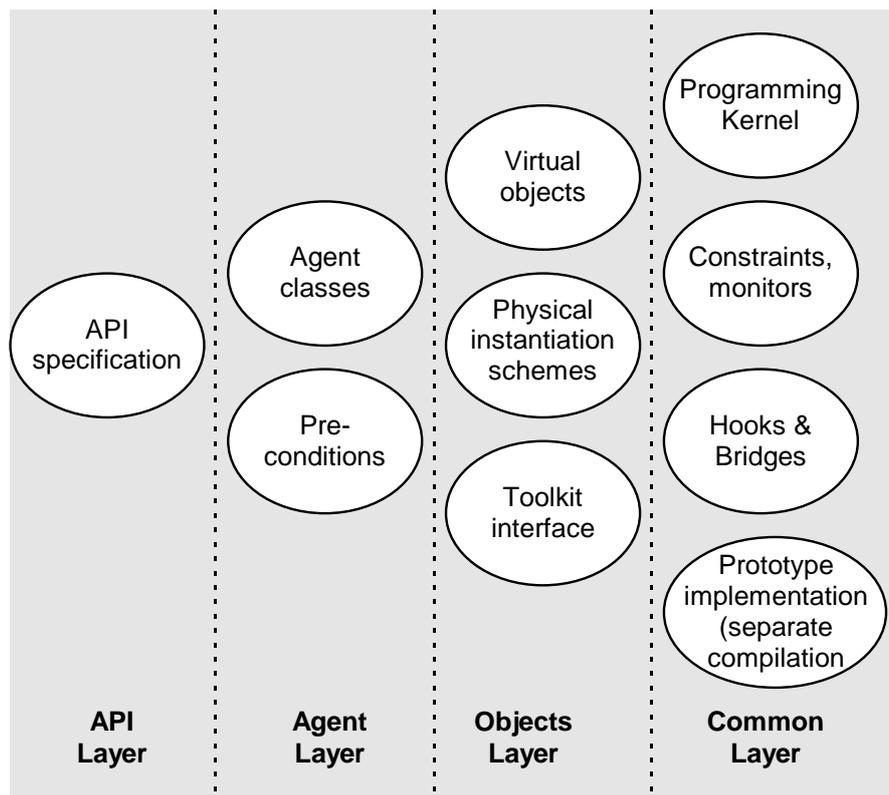


Figure 4.4 - The four layers of implementation constructs in the I-GET language.

4.2 METAPHOR DEVELOPMENT

Interaction metaphors are employed to increase learnability, ease-of-use, and user satisfaction, by supporting various real-world analogies within the interaction context [Carroll et al, 1988]. In the context of unified development, the advantages of metaphoric interaction are mainly [Stephanidis et al, 1997a]: (a) consistency of system behaviour expectations (from the user's point of view) with situations and phenomena from the real world; and (b) familiarisation with supported user-actions and corresponding system responses. The design of interaction metaphors requires in depth considerations of user attributes, as well as intended contexts of use. For instance the design of the windowing desk-top metaphor [Canfield et al, 1982] has been targeted towards able users, working in an office environment.

In this Section, we will firstly present the metaphor development methodology of the unified interface construction paradigm, primarily drawing an engineering path for the metaphor development process. This methodology is not for designing metaphors, but rather for organising the development process in a layered manner, enabling modular extensions and modifications, as well as re-implementation of metaphors for other user groups and contexts of use, than those for which it has been originally designed. Secondly, we will discuss the user-oriented nature of metaphor development, and show three representative examples of wrong metaphor use. Then, starting from the need of metaphoric interaction for diverse user groups, we will reveal the key role of top-level container interaction objects in providing alternative physical metaphoric representations. Finally, we will present a specific metaphor design case for non-visual interaction, which has resulted in an interface toolkit supporting fusion of alternative metaphors within the same interface instance, through flexible container interaction objects.

4.2.1 Metaphor Development Process

In the unified development paradigm, the metaphor development process is split in three distinct phases (see Figure 4.5): (i) *design* of metaphoric aspects, concerning the definition of the various metaphoric entities (e.g. electric device panel components like buttons and switches, desk-top surface visual structure, physical rooms), including relevant properties, behaviours and relationships, that are to be transformed in an interaction context; (ii) *realization* of a metaphor design, in which the construction of real interaction scenarios is to be performed, for all the aspects of the various designed metaphoric entities (e.g. media and modalities, interaction objects classes and associated attributes, dialogue design); and (iii) *implementation* of a metaphor realization, which deals with the provision of software libraries for building interfaces which are composed of dialogue elements taken from that particular metaphor realization.

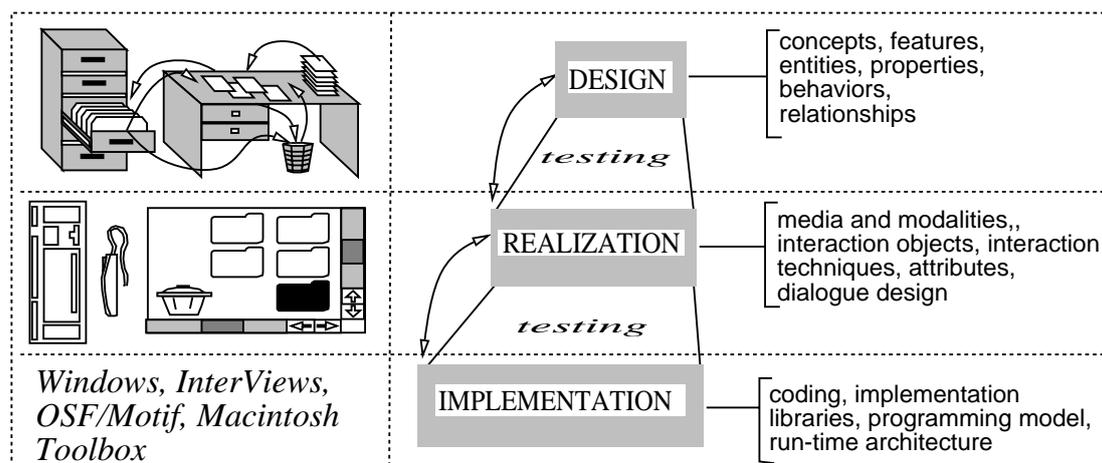


Figure 4.5 - Metaphor development stages in the unified interface engineering paradigm.

Following the previous scheme, modifications can be introduced to a particular level without necessarily affecting the levels above. For instance, various metaphor realizations can be constructed for a single design (e.g. Rooms entities with visual graphical, e.g. [Card et al, 1987], and non-visual realizations, e.g. COMONKIT [Savidis et al, 1995b] and AudioRooms [Mynatt et al, 1995]), while various implementations can

be built for a particular metaphor realization (e.g. various existing window-based toolkits, virtually implementing a common set of visual dialogue techniques - particular realizations of the desk-top metaphor).

4.2.2 The User-Oriented Nature of Metaphor Development

During the metaphor design- and realisation- stages, specific user attribute values need to be considered. Hence, the resulting metaphor design(-s) and realisation(-s) are directly associated to those user attribute values. One such representative example concerns the design and realisation of the desk-top metaphor, which is currently reflected in all windowing interactive environments. The original design had considered the needs of an "average" person working in an office and performing tasks primarily engaging information conveyed in sheets of paper. The resulting realization has been targeted towards sighted users, and has been based on the effective exploitation of the human-visual information processing capability.

It is argued that both accessibility, as well as interaction quality problems may arise when trying to deploy interaction metaphors across user populations other than those originally considered during the metaphor design and realisation stages. We will briefly present three cases, which, in the context of this thesis, are considered as being representative of wrong metaphor use; those will particularly emphasise the user-oriented nature of metaphoric interaction:

- *Windowing interaction for blind users.* This scenario is reflected in screen readers aiming to provide access to windowing applications by blind users. In this case, visual realisations of the desk-top metaphor are reproduced in a non-visual form. However, the metaphor realisation is even closer to sighted user needs, than the metaphor design itself, since specific visual interaction means are considered. In conclusion, fundamental entities (e.g. windows, icons, visual cues) and relationships (e.g. overlapping, spatial arrangement) in the desk-top metaphor, require considerable

further investigation in order to verify whether their reproduction in a non-visual form is meaningful.

- *Windowing interaction for children (preschool, early school).* Various software products for educational or entertainment purposes have been produced, targeted to children of the preschool or early school age, many of them working under popular windowing environments. Hence, the desk-top metaphor is directly employed for interaction. In this case, some of the common properties of windowing environments, such as concurrency of input actions, multitasking (e.g. many applications), intuitive data exchange among applications (e.g. copy / paste), direct manipulation and direct activation (e.g. drag and drop), etc., are mainly directed towards business / office tasks in a working environment. This problem has been recognised at an early point, leading to a new generation of *edutainment* software products, demonstrating a large amount of custom-made metaphoric interaction strategies like cartoons & animation, story telling, live characters and use of utopia.
- *Custom-made interaction controls in multimedia information systems.* This scenario falls in a category of software products commonly known as multimedia CD ROMs, providing mainly informative material in a hypertext manner. Currently, such products provide an amazing number of custom-made interaction controls, environment background and display styles. Unfortunately, unlikely edutainment software products, those applications reveal an extensive use of heuristically designed visual cues and effects, mainly coming from the real world; moreover, there are cases in which different interaction controls are employed within the same interactive application, even for performing the same tasks (e.g. activating operations, selecting from a list of items). It is common in such applications to move away from typical windowing interactive features, and employ many dynamic animated presentations and feedback techniques borrowed from TV advertisement spots.

4.2.3 The Key Role of Top-Level Containers in Metaphoric Interaction

Containers are those classes of interaction objects which may physically enclose arbitrary instances of interaction objects. In running interactive applications, those container object instances which are not enclosed within other containers are called *top-level containers*. For instance, windows providing interactive management facilities, which are not included within other windows, are called top-level windows.

When designing metaphoric interaction, there can be many real-world analogies which are *transferred* in the interaction domain; hence, practically, multiple distinct metaphors may be combined. For example, in windowing applications, the following interaction object classes are typically met, each representing a specific real-world analogy:

- Windows - *sheets of paper*.
- Push buttons, sliders, potentiometers, gauges - *electric devices*.
- Check boxes - *form filling*.
- Menus - *restaurant*.
- Icons - *signs*.

Naturally, the real-world physical relationships as such are broken, when containment relationships are designed (e.g. none would expect to see an electric button on a sheet of paper in the real world, while push buttons are normally embedded within windows). The effect of such containment relationships is that interaction metaphors are embedded at various levels in interactive applications. Work has been already carried out before, investigating the design aspects of embedded interaction metaphors [Carroll et al, 1988]. In the context of the unified interface development paradigm, we have particularly focused on the identification of those interactive entities which play a key role in providing the overall metaphoric nature of an interaction environment. This research effort has been carried out with the following spirit:

In unified interface development, diverse users may require different interaction metaphors so as to accomplish high quality of interaction. If we are able to detect those entities largely affecting the overall metaphoric environment “look and feel”, then we may only need to provide different metaphoric representations for those entities, in order to derive alternative metaphoric environments. This would potentially alleviate the overhead of designing from scratch alternative metaphoric artefacts, since only specific changes on those design artefacts, regarding the important metaphoric entities, would be required.

The resulting methodological framework to address the above issue is based on the role of top-level containers. It is argued that *the overall interaction metaphor is characterised by the metaphoric properties of top-level containers, while all embedded interaction objects are physically projected within the interaction space offered by the top-level containers*. Driven from this principle, two specific research efforts have been carried out, one following the other: (a) a non-visual toolkit called COMONKIT [Savidis et al, 1995b] has been designed and implemented, as part of the GUIB Project [GUIB Project, 1995], providing a single top-level container with Rooms-based interaction, and many standard interaction object classes like “menu”, push button”, etc.; and (b) a non-visual toolkit called HAWK [Savidis et al, 1997c] has been designed and implemented, as part of the ACCESS Project [ACCESS Project, 1996], providing a generic container object, capable of realising various metaphoric representations, as well as various conventional interaction objects classes (like in COMONKIT). The interaction design for the HAWK toolkit will be discussed in the next Section; this toolkit has been effectively utilised in building real-life applications, like the non-visual interface of a dual electronic book [Petrie et al, 1996], as well as the non-visual dialogue patterns for a unified implementation of a user-adapted Web-browser [Stephanidis et al, 1997b].

The key role of top-level containers has been identified on the basis of various dialogue artefacts, for which designers had to characterise the overall interaction metaphor. The most representative collection of those dialogue scenarios is provided in Figure 4.6. This collection has been presented to the participants of a tutorial in

HCI International'97 conference, San Francisco, California, USA, [Stephanidis et al, 1997a]. In all of the above cases, it has been argued that embedded objects could not affect the overall interaction metaphor.

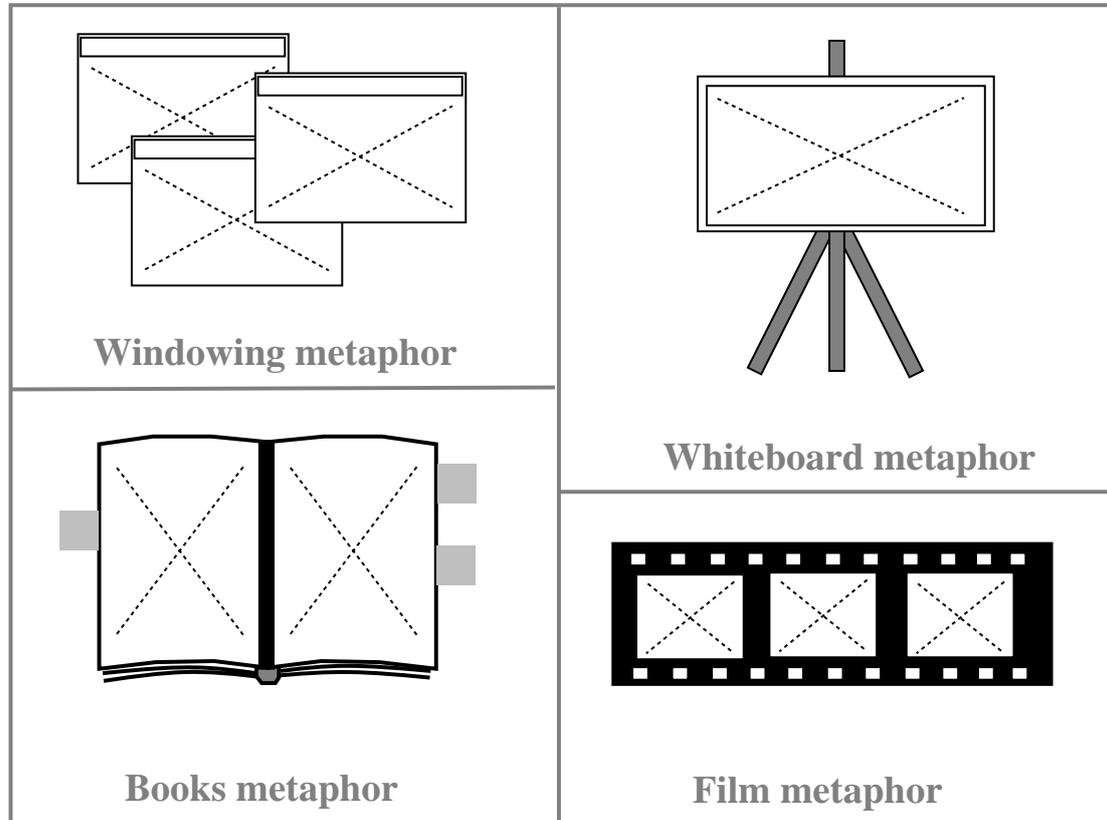


Figure 4.6 - The four design scenarios demonstrating how top-level containers affect the environment metaphoric properties.

4.2.4 A Metaphor Development Case

This effort has been carried out in the process of providing implemented non-visual interaction elements, within the ACCESS Project [ACCESS Project, 1996], so that developers could assemble the non-visual dialogue components of unified interactive applications. In this context, it has been quickly acknowledged that:

“providing the right metaphor for blind people opens new opportunities for interaction, in the same manner that selecting the wrong metaphor drives developments in the wrong direction.”, [Savidis et al, 1997c],

and,

“the metaphor design and implementation process requires huge design, implementation, testing and evaluation resources. Moreover, it is hard to choose metaphoric representations universally acceptable to all users, which also address all application domains.”, [Savidis et al, 1997c].

Hence, we had to avoid hard-coding a single particular metaphor within the software libraries which had to be produced, since the overhead for modifications would be hard to practically manage. As a result, starting from the principal role of container objects, the HAWK toolkit [Savidis et al, 1997c], [Savidis et al, 1997h], has been designed, which provides a single container, supporting alternative metaphoric representations.

The container class in the HAWK toolkit provides various presentation and dialogue attributes through which alternative metaphoric non-visual styles can be derived, by appropriately combining messages and sound feedback: (a) synthesized speech message (speech message to be given when the user "focuses" on the object); (b) Braille message (message displayed on Braille device when the user "focuses" on object); (c) "on-entry" digitised audio file (to be played when the user focuses the object); and (d) "on-exit" digitized audio file (to be played when the user "leaves" the object).

In Figure 4.7 the assignment of specific values to the container presentation attributes is shown, in order to derive alternative metaphoric representations; three container instances, realising “books”, “desk-top” and “rooms” metaphors are defined. In the case of non-visual interaction, it has been relatively easy to design such parameterised metaphoric representations, due to the simplicity of the output channels (i.e. audio, speech and Braille). This approach has been validated in the context of the ACCESS

Project [ACCESS Project, 1996], both with respect to its usability as an engineering method, as well as with respect to the usability of the produced interfaces [Savidis et al, 1997].

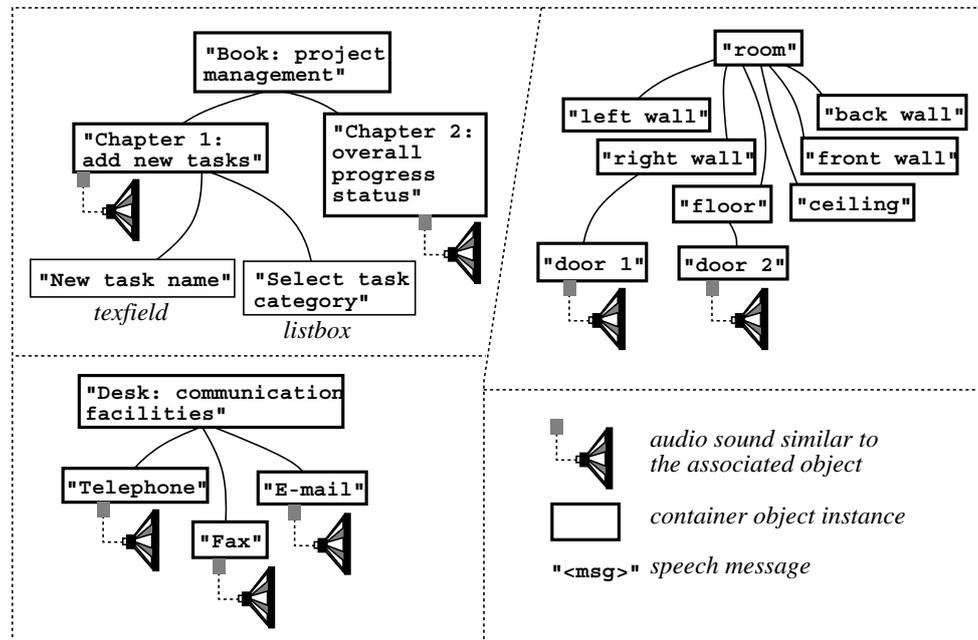


Figure 4.7 - Supporting alternative metaphoric representations for container instances in object hierarchies.

Trying to apply the same implementation technique in graphical interfaces, i.e. implementing graphical containers which provide attributes for controlling metaphoric properties, is likely to require a huge development overhead (because highly parameterised graphical structures would be required). Hence, in case of graphical interfaces, since distinct metaphoric top-level containers have to be explicitly hard-coded, it is important that extensive design and usability evaluation cycles are performed, before providing those containers in a target implementation form.

4.3 TOOLKIT INTEGRATION

We consider as toolkits all kinds of software libraries providing implemented interaction elements. A given interface development tool is considered to support toolkit integration, if it allows importing of *any* particular toolkit, so that *all* interaction elements of the imported toolkit(-s) are subsequently “exposed” (i.e. made available) within the original interaction building techniques of that particular given interface tool. For instance, if an interface builder providing graphical construction techniques supports toolkit integration, then, by integrating a particular toolkit (which supplies interaction objects with a specific “look and feel”) the original interactive graphical design facilities should directly enable manipulation of those interaction object classes. In this definition, we do not assume any particular interface construction method for interface tools. Hence, toolkit integration could be supported by: programming-based tools, interactive interface builders, state-based tools, event-based tools, demonstration-based tools, interface implementation 4GLs (4th Generation Languages), etc.

The need for importing toolkits is evident in cases that the interaction elements originally supported by the particular interface development tool do not suffice. This is a possible scenario if interface development for diverse user groups needs to be addressed. For instance, in the context of Dual Interface development [Savidis et al, 1995a], where interfaces concurrently accessible by sighted and blind users need to be constructed, non-visual interaction techniques are required, together with typical visual graphical interaction elements. Existing windowing toolkits do not supply such interaction techniques; hence, integration of special-purpose non-visual interaction toolkits, such as COMONKIT [Savidis et al, 1995b], or HAWK [Savidis et al, 1997c] is necessitated. In the context of the User Interfaces for All objective, such scenarios are likely to emerge, since developing for diverse user groups is of primary importance.

4.3.1 Previous Related Work

The toolkit integration capability implies that interface tools supply mechanisms which are made available to developers (i.e. to be utilised *after* the tool-product is launched). Currently, a very small number of interface tools supports this notion of platform connectivity. The first interface tool which provided comprehensive support for toolkit integration has been the SERPENT UIMS [Bass et al, 1990], where the toolkit layer has been called lexical technology layer; the architectural approach developed in the SERPENT UIMS revealed key issues in toolkit interfacing. The HOMER UIMS [Savidis et al, 1998], which has been developed to facilitate the construction of Dual User Interfaces, also supported toolkit integration, by providing a powerful integration model, general enough to enable integration of non-visual interaction libraries, apart from traditional visual windowing toolkits.

We should make an explicit distinction among toolkit integration requirements and the multi-platform capability of certain toolkits. In the latter case, a single toolkit is provided with multiple fixed implementations across different OS platforms, already made available when the toolkit product is released (i.e. multi-platform toolkits like YACL, Amulet, XVT, and JAVA AWT library - a review can be found in [Guinan, 1997]). In the former case, a tool is made open, expecting that tool users will take advantage of the well documented functionality for connecting to arbitrary toolkits.

4.3.2 Implementation Requirements

There are two categories of implementation requirements, each judging the extent to which toolkit integration is supported by a given interface tool: (a) *minimalistic requirements*, which characterise a particular tool with respect to whether it supports some degree of openness, so that interaction elements from external (to the given interface tool) toolkits can be utilised (subject to some particular implementation restrictions); and (b) *maximalistic requirements*, revealing the full range of functional capabilities regarding toolkit integration, which, when met by interface tools, turn

toolkit integration to a practically simple task (also, various “parameters” are likely to be supported, concerning the way integration may be carried out in practice). The detailed definition of these two categories of functional requirements for toolkit integration follows.

4.3.2.1 Minimalistic Requirements

- Ability to *link / mix* code at the software library level (i.e. combining object files, linking libraries together).
- Support for documented *hooks*, in order to mix at the source code level (i.e. calling conventions, type conversions, common errors and compile conflicts, linking barriers).

If the minimalistic requirements are satisfied, it will be made possible to combine together software modules which utilise interaction elements from different toolkits. The question is whether existing interface tools support the minimalistic requirements. The answer to this question is both affirmative, and negative.

Affirmative, in those cases, where the toolkit to be integrated provides different categories of interaction elements with respect to the interface tool being used (providing the integration service). For instance, assume that the interface tool being used is a programming-based toolkit of windowing interaction elements. Then, if audio-processing functionality for auditory interaction is to be imported and combined with this particular tool, potential conflicts can be easily resolved.

Negative, in those cases, that the imported toolkit supplies similar categories of interaction elements with respect to the interface tool being used. For example, trying to combine various libraries of windowing interaction elements, from different vendors, leads to serious conflicts (e.g. WINDOWS object libraries from different vendors, Xt-based toolkits like OSF/Motif and Athena widget set). The primary

source of such conflicts is *name collision*, at the binary library level, due to commonly implemented constructs within such toolkits like “object”, “window”, “button”, “initialize_toolkit”, “close_toolkit”, “event_handler”, etc. In all such cases the interface toolkits reserve exclusively some names (like those previously mentioned) for particular software library entities, and they do not allow other implemented entities to be registered with those specific identifiers.

Additionally, in cases where the various toolkits to be concurrently utilised share common libraries (e.g. like Xt / Xlib based toolkits), it is not allowed to call particular services (e.g. functions) more than once. Typically, such “problematic” services provide some start-up registration facilities, like opening a connection with a window manager. In all cases studied, multiple calls to such services caused run-time errors, thus practically disabling initialisation sequences to be executed for more than one toolkits (in a client utilising more than one toolkits). Because of this behaviour, we call this family of conflicts as the *exclusive registration* conflict.

4.3.2.2 Maximalistic Requirements

- Well behaved and well documented compilation / translation and linking cycles for interfaces utilising the integrated toolkit(-s); this applies to all types of interface building methods, not only programming-oriented tools.
- Single implementation model made possible for all integrated toolkits; for instance, when the target interface tool provides visual construction methods, then the same facilities should allow manipulation of interface elements from all integrated toolkits.
- Ability to change aspects of the programming interface (i.e. the programmable view) of each imported toolkit. For instance, assume that target programmers are familiar with a particular programming style. Then, when importing software toolkits, it might be necessary to alter some aspects of their original programming interface (e.g. naming conventions, programming model), without affecting the

underlying implementation, so that it can approach that specific programming style.

- Resolve the following types of problems which appear when trying to combine multiple toolkits together: (i) *language conflicts*, since not all software libraries are made available through the same programming language; (ii) *compile conflicts*, since, even when the same (or compatible) programming languages are supported among toolkits, collision of data-types, variables, constants, etc., is a common phenomenon; (iii) *linking conflicts*, mainly due to name collision at the binary library levels; and (iv) *execution conflicts*, due to the exclusive registration phenomenon.
- Ability to import any type of interface toolkit, irrespective of the style of interaction supported (e.g. windowing toolkits, auditory / tactile toolkits, VR-based interaction toolkits).
- Ability to combine toolkits together for creating cross-toolkit object hierarchies; this is defined as the *toolkit interoperability* requirement.

Currently, there is no single interface tool known which meets all the maximalistic requirements listed above. Regarding the notion of single programming interface, multi-platform toolkits already provide adequate support by means of a fixed programming layer. Only UIMS tools like SERPENT [Bass et al, 1990] and HOMER [Savidis et al, 1995a] supply adequate support for toolkit integration by enabling the establishment of developer-defined programming interfaces on top of software toolkits. Such a capability is also supported by the I-GET UIMS, as it will be explained later on, and is also provided by the PIM tool [Savidis et al, 1997a], a spin-off of the I-GET UIMS, which has been built for providing general purpose toolkit integration services.

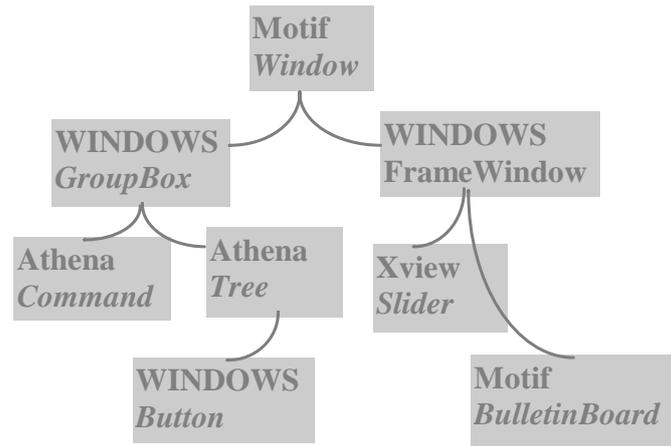


Figure 4.8 - An example of a cross -toolkit object hierarchy (WINDOWS, Motif, Athena and Xview objects are mixed).

When a single programming interface is supported for all platforms, one important question concerns the “look and feel” across those platforms. There are three alternative ways to cope with the “look and feel” in this case:

- *Adopt the native interaction controls* as they are provided by each underlying platform;
- *Mimic the native controls* of each platform;
- *Provide custom-made interaction elements* for all target platforms.

Regarding toolkit interoperability (see Figure 4.8), only the Fresco User Interface System [X Consortium, 1994] is known to support cross-toolkit hierarchies, where mixing of InterViews-originated objects with Motif-like widgets is facilitated.

It should be noted that even though elements from different toolkits may be combined, possibly employing a different “look & feel”, consistency is not always damaged. In Figure 4.9, a scenario of mixing a windowing toolkit with a Books-toolkit is outlined, requiring a cross-toolkit object hierarchy at the implementation level. The advantages of mixing multiple toolkits are more evident in case that the combined toolkits offer container objects with different metaphoric representations

(like the example of Figure 4.9), thus practically leading to the fusion of alternative metaphors. Various innovative dialogue artefacts may emerge, leading to the potential enhancement of the interface quality, an effect contributing towards the objective of “User Interfaces for All”.

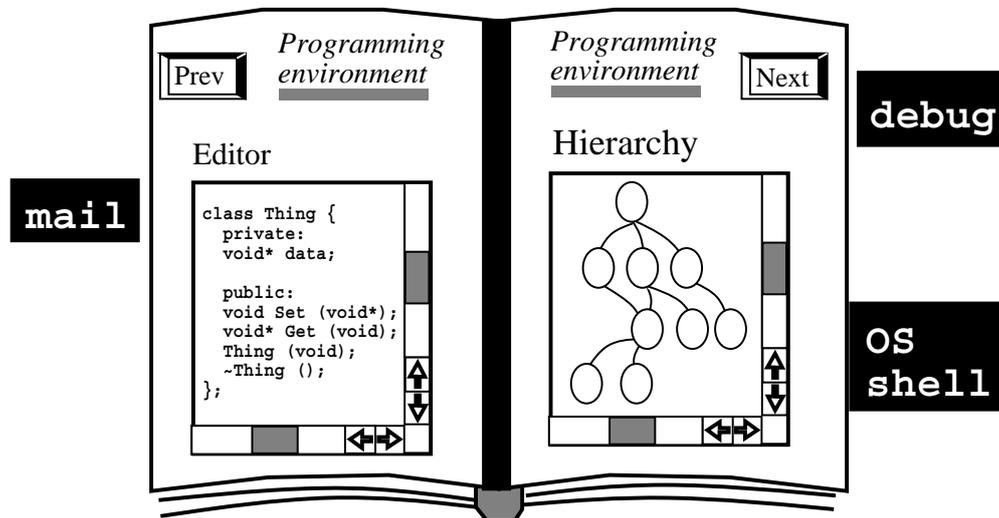


Figure 4.9 - A dialogue artefact for mixing container objects (coming from different toolkits) which comply to different interaction metaphors.

4.3.3 Toolkit Integration Support in the I-GET UIMS

The I-GET UIMS provides toolkit integration facilities which satisfy the maximal implementation requirements, as they have been previously defined. During the design of the I-GET toolkit integration mechanism, two key issues had to be addressed: (a) how developers view and manipulate, through the I-GET language, the imported interaction elements; and (b) how the run-time connection with the underlying toolkit software library(-ies) is established. Below we briefly discuss how these two top-level issues have been addressed, while outlining the toolkit integration strategy of the I-GET UIMS.

The first step in toolkit integration is the *toolkit interface specification*. In this stage, the structure and type of the various interaction elements to be integrated is specified.

Such a specification is characterised as an “interface” specification, in a programming sense, since it does not encompass any type of executable statements or constructs (analogous to CORBA Interface Definition Language). The resulting specification plays a two fold role: (i) it provides the programming interface of imported interaction elements in the I-GET language, encompassing all the necessary information so that developers may effectively utilise those elements, in an analogous manner that function prototypes include enough information for calling functions in C / C++; and (ii) it is parsed by the I-GET compiler, to generate special purpose C++ software modules, which will have to be explicitly employed in developing a corresponding toolkit server, the latter playing the role of the run-time translator between the I-GET developed interfaces and the imported toolkit library.

The second step is the actual *toolkit server development*, in which the run-time link between the I-GET developed interfaces and the underlying toolkits is drawn. The I-GET tool automatically generates template structures, from toolkit interface specifications, around which, toolkit servers can be easily built.

4.3.3.1 Toolkit Interface Specification

The toolkit interface specification kernel of the I-GET language had to primarily exhibit two distinctive properties: (a) it should not reflect any particular interaction style or metaphor, such as windowing interaction; and (b) it should allow integration of all sorts of lexical interaction constructs met in programming toolkits. While aiming towards meeting these two requirements, it has been considered as more appropriate to firstly formulate an appropriate meta-model, falling in the programming domain, of lexical interaction elements, which would be potentially applicable to all type of interface toolkits. Then, on the basis of this meta-model, we could instantiate an appropriate language kernel suited to the I-GET language. The significant potential of such a meta-model is that it could be mapped to various syntactic forms in different languages; actually, this exercise has been practically taken in the context of the PIM tool [Savidis et al, 1997a], where the toolkit meta-

model defined in the context of this thesis has been also mapped to appropriate C++ language constructs.

4.3.3.1.1 Generic Toolkit Metamodel

The designed meta-model, called the *Generic Toolkit Metamodel* (GTM), constituted the basis of the toolkit interface specification formalism; according to the GTM, interface toolkits provide interaction elements which fall in any of the following three fundamental classes: *objects*, *input events* and *output events*.

4.3.3.1.1.1 Objects

Objects have an arbitrary number of typed *attributes* (like "x", "font", "text", etc) and an arbitrary number of *methods* (like "Selected", "Pressed", "FocusIn", etc). Methods typically characterize what the user is able to do with an object (e.g selecting an option, pressing the button). Apart from conventional interaction objects (e.g. "Buttons", "Textfields"), objects can represent graphic interaction elements (e.g. "Lines", "Circles"). In this case, the attributes of an object will correspond to the parameters of the particular graphic element (e.g. "x", "y", "radius"), while methods will characterize the interactive manipulation facilities (e.g. "Picked", "Moved", "Rotated", "Stretched"). The model does not restrict the physical nature and target metaphor for interaction objects; for instance a "Room" object [Savidis et al, 1995b] can be modelled, with its various attributes (e.g. "soundOnEnter", "soundOnLeave", "tactileOn") and methods (e.g. "Entered", "Exited") directly expressed.

4.3.3.1.1.2 Input events

Input events have an arbitrary number of typed parameters, carrying event-specific data, and asynchronously occur in the context of objects. Events may concern: device-oriented input (e.g. key presses, mouse moves), notifications for window management events (e.g. exposure / Xlib, repainting / WINDOWS, window destruction / WINDOWS) or navigation-specific events for alternative metaphors (e.g. "glove pos", "glove obj", "ring left voice command", etc, for a 3D auditory ring navigation space

supporting 3D hand pointing [Savidis et al, 1996], client specific processing (i.e. user events for WINDOWS / Xlib) that may generate high-level events (e.g. a gesture recognizer generating gesture events [Zhao, 1993]), etc.

4.3.3.1.1.3 Output events

Output events have an arbitrary number of *output parameters* (necessary for the toolkit to perform an output event) and an arbitrary number of *input parameters* (values possibly returned from the toolkit to the caller, after an output event has been served). Output events are practically functions of the form $y^n=f(x^m)$, where x^m is the list of output parameters, while y^n is the list of input parameters. Output events may concern: procedural graphics primitives and drawing facilities (e.g. "POINT_ABS_2" , "VIEWPORT_2", "LINE_REL_2" - [Foley et al, 1983]), toolkit functions for managing interaction objects (e.g. "RealizeWidget" / Xt, moving / hiding / re-displaying / destroying objects), toolkit resource management (e.g. "CreateGC" / Xlib, font management, bitmap manipulation), window management or navigation functions (e.g. changing listener window / Xlib, "ring turn left" / "ring turn right" -[Savidis et al, 1996]), etc.

```
lexical (Xaw);
lexical Button (Xaw) [
    method Pressed;
    int x=0, y=0;
    int width, height;
    string label;
]
inputevent MouseMoved (Xaw) [
    int x, y;
]
outputevent GetBitmap (Xaw) [
    out:
    objectid obj;
    int x, y, width, height;
    in:
    int bitmapid;
]
```

Figure 4.10 - Definition of lexical interaction elements in toolkit interface specification (example from Xaw/Athena toolkit integration).

4.3.3.1.2 Toolkit Interface Specification Kernel

The I-GET language allows multiple sets of imported interaction elements to be manipulated at once. All imported interaction elements which physically belong to the same toolkit should be logically grouped together under the umbrella of a common toolkit name. In Figure 4.10, issuing the name of a toolkit called **Xaw**, corresponding to the Athena widget set, is provided; the toolkit name must be unique and it can be arbitrarily chosen. After defining one or more toolkit names, various interaction elements may be defined; every interaction element has to be explicitly associated to a particular toolkit name already defined. For instance, in Figure 4.10, an object called **Button**, an input event called **MouseMoved** and an output event named **GetBitmap** are defined, all associated with the **Xaw** toolkit. The specification body of interaction elements is provided between an opening [and a corresponding closing] square bracket. The I-GET language supports various data types for object- and input / output event- parameters, such as enumerated, structures and types synonyms, as well as various variable parameter classes, like pointers and arrays.

Declaring instances of imported objects	<code>lexical(Xaw) Box box; lexical(Xaw) Button button : parent={box};</code>
Calling output events	<code>int bid; out(Xaw).GetBitmap({box}, 0, 0, 10, 10, &bid);</code>
Artificial input event broadcasting	<code>in(Xaw).MouseMoved({box}, 25, 37);</code>
Accessing object attributes	<code>{button}.x={box}.x+10; {button}.label="Press";</code>
Implementing methods	<code>method {button}.Pressed [{button}.label+="!"; // add a '!' when pressed.]</code>

Figure 4.11 - Supported scenarios of use of imported interaction elements for interface development with the I-GET languages.

The imported interaction elements may be directly employed in interface development, through the facilities offered by the I-GET language. In Figure 4.11, the supported scenarios of use are illustrated. Object instances may be declared, structuring arbitrary object hierarchies. Since multiple sets of interaction elements may be defined, the toolkit name should explicitly qualify the object class in instance declarations. Arbitrary implementations may be added to methods of object instances, thus supporting multiple call-back registrations. Output events may be called in a

syntax similar to function calls; the `out(<toolkit name>)` prefix is syntactically necessary to indicate that an output event from a specific toolkit is to be called.

```

eventhandler {box}  [
    bool locked=false;
    int x1,y1, x2, y2;
    (true) MousePressed [
        locked=true;
        x1=x2=MousePressed.x;
        y1=y2=MousePressed.y;
        Draw(x1,y1,x2,y2);
    ]
    (true) MouseMoved [
        if (locked==true) [
            Hide(x1,y1,x2,y2);
            x2=MouseMoved.x;
            y2=MouseMoved.y;
            Draw(x1,y1,x2,y2);
        ]
    ]
]

```

Figure 4.12 - An event handler specification for rubber-banding definition of a line.

Input events may be artificially generated, supporting both local- (as in [Hill, 1986]) as well as non-local- event broadcasting (through the explicit resolution of the target object instances - e.g. `{box}` in the example of Figure 4.11). Apart from artificial event generation, the I-GET language supports the specification of event handlers, providing a model which forms an extension to Event Response Language (ERL), originally introduced in [Hill,1986].

The extended features are: (i) arbitrary expression-based preconditions to event blocks, in comparison to simple true / false boolean flags in ERL; (ii) ability to fire multiple event blocks, if they all match the same event class in the event queue, adding more concurrency to the original Event Response System (ERS) model [Hill, 1986]; and (iii) extension for non-local event broadcasting capability from within event blocks. In Figure 4.12, a simplified version of an event handler for interactive definition of lines via rubber-banding techniques is provided. The `Draw()` and `Hide()` functions are user-defined and are omitted for clarity; for instance, the

`Hide()` function uses imported line drawing output events to draw lines with the window background colour.

4.3.3.1.3 Toolkit Interface Specification Strategies

The toolkit interface specification through the I-GET language realizes a particular programming "view" of the original toolkit functionality; hence, it actually specifies the programming interface of imported toolkits within the I-GET language. From the "distance" between the specified interaction elements, during the toolkit integration process, and the specific programmable structure of elements, as originally met within the imported toolkits, five alternative strategies for toolkit interface specification are distinguished (combinations are also made possible). A brief description of those five specification approaches follows; those have been originally introduced in the context of the PIM tool [Savidis et al, 1997a], supplying toolkit integration services.

4.3.3.1.3.1 *Mapping*

The original naming conventions, as well as the number and structure of elements, are preserved during toolkit interface specification. The resulting specification effectively provides the original toolkit elements, though via the I-GET language programming model. Mapping is appropriate for quickly integrating a particular toolkit; further modifications on the programming interface resulting from such an integration process, in the domain of the I-GET language, can be subsequently applied adopting various approaches to be discussed next.

4.3.3.1.3.2 *Renaming*

Within toolkit interface specification, the original toolkit naming conventions for interaction elements can change. Even with such simple modifications, there are a number of practical advantages like: (i) toolkit elements and their attributes / parameters may become more understandable; or (ii) naming conventions from toolkits already known by end-programmers may be employed.

4.3.3.1.3.3 Simplification

This is a combined application of mapping and renaming, where the number of specified elements and also their associated attributes / parameters, is purposely reduced. Emphasis is put on the ease of use (i.e. programming) of the resulting programming interface.

4.3.3.1.3.4 Transformation

This is the general case, in which virtually all aspects of toolkit elements (i.e. naming conventions, number and structure of elements) can be changed during specification. By applying transformation, the distance between the toolkit elements and the resulting toolkit interface specification may be considerably increased, and in some cases (as it will be discussed later on) the toolkit server development task (for connecting the I-GET language layer with the original toolkit) may become more complicated.

4.3.3.1.3.5 Generalization

This approach leads to a toolkit interface specification which is appropriate for integrating multiple toolkits. It is a special case of transformation, while due to the fact that the resulting programming interface is applicable for manipulating the interaction elements of more than one toolkits, it practically constitutes a virtual toolkit [Myers, 1995] (it will have to be connected to multiple target toolkits, via the implementation of multiple toolkit servers).

4.3.3.1.3.6 Examples of the Various Toolkit Interface Specification Strategies

Some specification examples will be provided, demonstrating the various specification strategies in practice. Most of these examples, which are provided in Figure 4.13, will adopt more than one of the previously defined strategies.

<code>lexical (Xaw);</code>	<code>lexical ListBox (W95) [method Selected;</code>
-----------------------------	---

<pre> enum ShapeStyle=[Rect, Oval, Ellipse, RoundedRect]; lexical Command (Xaw) [int x, y, width, height; string font, fg, bg; ShapeStyle shapeStyle; method Activated;] lexical (W95); lexical Button (W95) [method Pressed; string title;] lexical TextField (W95) [method Edited; string text;] </pre>	<pre> method Unselected; string whichItem; int itemIndex; string* items; bool* selectStatus;] lexical (ViKit); lexical Button (ViKit) [method Pressed; string label; int x, y, width, height; string fg, bg, bordColor; int bordWidth;] inputevent KeyPress (ViKit) [int presCount; string key;] outputevent GetGraphics (ViKit)[out: objectid obj; in: int gid;] </pre>
---	---

Figure 4.13 - (1) Mapping, Renaming and Simplification for Xaw/Athena widget set; (2) Simplification for interaction controls from the Windows object library; (3) Transformation and Simplification in the Windows object library; and (4) Generalisation for production of a virtual toolkit (called ViKit) in the integration of Xaw/Athena and Windows object library.

In Example 1 of Figure 4.13, the name of the object class (i.e. **Command**), as well as the types and names of the attributes **x**, **y**, **width**, **height**, **sensitive**, **label**, and **shapeStyle** are preserved from the original Xaw/Athena class (i.e. Mapping). Some callback classes like **xtnDestroy** and **xtnCreate**, supported in the corresponding Xaw/Athena widget classes, have been removed (i.e. Simplification). The **Activated** method corresponds to the **xtnSelected** callback class in Xaw/Athena (i.e. Renaming).

In Example 2 of Figure 4.13, the specified **Button** class has the same name in Windows object library, while the **TextField** class corresponds to the **EditControl** toolkit class. A single attribute and a single method have been defined

for each of the two classes; considerable reduction of attributes, as well as renaming for the remaining attributes and methods has been applied (i.e. Transformation and Simplification).

In Example 3 of Figure 4.13, a typical example of transformation is shown. In Windows object library, a list box object supports the selection of multiple items by the user (as opposed to the **List** widget class in Xaw/Athena, where a single user-selection is only made possible). The original programming interface for the **ListBox** class has been transformed and simplified: **items** concerns the list of string items initially supplied, **Selected** and **Unselected** indicate the callback lists for selection and "un-selection" (i.e. pressing an already selected list item), while **whichItem** and **itemIndex** return the content and the index of the selected / unselected item respectively. The attribute **selectStatus** is an array of flags holding the selection status for each of the list items. In the context of the ACCESS project [ACCESS Project, 1996], programmers have found much easier [Savidis et al, 1997i] this programming interface for multiple-choice list boxes, than the original one in Windows object library.

Finally, in Example 4 of Figure 4.13, the Generalization strategy is demonstrated. The resulting virtual programming interface is named **ViKit** (the name can be arbitrary). One object class is specified, called **Button** (i.e. **Button** for Windows object library, **Command** for Xaw/Athena), which has one method named **Pressed** (Windows and Xaw/Athena methods have been renamed). The **GetGraphics** output event "unifies" the **CreateGC** Xlib call, and the **GetDC** Windows call).

4.3.3.1.4 Minimal Object Modelling

Currently existing toolkits provide programming structures of interaction objects, like dedicated classes or opaque data types, which realize a "flat" modeling of the various object properties. For instance, widgets which correspond to interaction object entities in Xt-based toolkits, provide a linear set of attributes like "x", "y", "foreground", "background", "accelerators", "font", "sensitive", etc. Such attributes may be further

classified as *presentation* attributes (e.g. "x", "y", "foreground", "background", "font"), *input style* attributes (e.g. "accelerators") and *behavior* attributes (e.g. "sensitive"). Unfortunately, this approach of explicitly supporting property categories is lacking. Also, in all known toolkits the available interaction objects typically support hard-coded behaviours and presentation structures, without providing programming control on lexical interaction properties. As a result, well known powerful models of lexical input behaviours, like interaction tasks and interaction techniques [Foley et al, 1984], are not practically supported within the programming models of interaction objects offered by existing toolkits, since only a limited set of alternatives (i.e. one / two interaction techniques) are usually provided.

In this context, it is considered as a promising strategy to pursue the establishment of more sophisticated programming models for interaction objects, clearly categorizing the various properties of interaction objects within appropriately defined corresponding property classes (e.g. input technique, interim feedback, output structure, behaviour attribute, presentation attribute). Such efforts would promote, for instance, the modular extension of the lexical interaction facilities of interaction objects by providing implemented alternatives for the various property classes (e.g. various input techniques, alternative approaches to interim feedback), potentially leading to one possible implementation framework for *toolkit augmentation*.

Additionally, the provision of more sophisticated programming models of interaction objects, apart from offering better programming control of interaction properties to interface developers, is also considered as particularly beneficial for other purposes. For instance, interface design assistants may employ those sophisticated interaction object programming models in order to draw more targeted interface adaptation / design decisions regarding the lexical level [Akoumianakis et al, 1996]. Starting from the above remarks, a specific programming template for organising interaction objects properties has been defined, called *minimal object modelling*.

Output Techniques
<p>Concern the various methods supported for presenting objects of that particular object class. For example, a “menu” object in a windowing toolkit would exhibit the following presentation techniques for the list of displayed options:</p> <ul style="list-style-type: none"> • <i>Vertical</i> arrangement. • <i>Circular</i> arrangement. • <i>Sequential</i> presentation of options. • Options presented as <i>stacked</i> cards.
Attributes per Output Technique
<p>These are the various output parameters for each particular output technique. For instance, assuming a circular topology of “menu” options, relating to the previous example, the following attributes may be supported:</p> <ul style="list-style-type: none"> • Circle parameters (<i>a</i> and <i>b</i>, assuming an ellipse in the general case). • Flag for <i>auto-splitting</i> the circle in sectors for each option. • <i>Foreground / background</i> colours.
Input Techniques
<p>These concern the various ways through which the user may provide input and interactively manipulate the particular object instance. Input techniques are also related to the notion of interaction tasks, as defined in [Foley et al, 1984]. Input techniques are in many cases directly dependent on particular output techniques. Hence, the categories of properties in minimal object modelling are not always orthogonal to each other (i.e. combinations may be subject to toolkit implementation restrictions). Examples of input techniques, for a “menu” object are:</p> <ul style="list-style-type: none"> • <i>Direct</i> selection, <ul style="list-style-type: none"> • Via mouse; • Via an eye gaze device; • Through speech-input; • Through short-cuts. • <i>Indirect</i> selection, <ul style="list-style-type: none"> • Auto scanning;; • Manual scanning; • Sequential presentation of options enabling selection; • Focus on desired option and selection.
Attributes per Input Technique
<p>These are the available parameters to fine-tune the way input is made possible for a particular input technique. Such attributes are also split into two sub-categories: presentation attributes, which concern the various appearance parameters for a particular input technique, and behaviour attributes, related to the way in which an input technique can be fine-tuned, to deviate its behaviour. For instance, considering the case of indirect selection and auto scanning, the following attributes are identified:</p> <ul style="list-style-type: none"> • Option <i>feedback method</i>, <ul style="list-style-type: none"> • Highlight; • Shape outline; <ul style="list-style-type: none"> • Border width; • Line style and colour. • Scanning <i>time interval</i>, • Scanning <i>direction</i>, • Always forward and recycle; • Always backward and recycle; • Start with forward and change direction on recycle. <p style="text-align: right;"> Presentation Behaviour Behaviour </p>

Figure 4.14 - Minimal Object Modelling - definition and explanations.

This model emphasises the role of the various programming object attributes play in affecting the object's interactive features, clearly bringing together attributes having a similar purpose, by organising them into appropriate categories. Its application is particularly suited for toolkit interface specification, since at that stage, maximum flexibility is provided in altering the original programming model of interaction objects.

The model, which is illustrated in Figure 4.14, provides only a template which has to be adopted, suggesting two main categories of object attributes; it is not prescriptive with respect to the type, number, names and nature of specific interaction properties. In this sense, it is considered to minimally affect the toolkit interface specification process, since it only provides logical grouping of attributes, rather than introducing new attributes not originally supported by the imported toolkits. The minimal object modelling still follows the Generic Toolkit Metamodel, being one particular instantiation of the latter.

4.3.3.2 Toolkit Server Development

The toolkit server plays the role of the intermediate "translator" between the Dialogue Control module, of the I-GET run-time architecture, and the particular imported toolkit. The toolkit server has a twofold role: (i) it receives requests from the Dialogue Control, corresponding to object instantiations, attribute modifications, and output events, that the interface client program (developed in the I-GET language) will normally perform, and it serves such requests by calling the original toolkit functionality; the "physical" interface is created and maintained locally within the toolkit server; and (ii) it sends messages back to the Dialogue Control module regarding input events, method notification, and attribute modification, as a result of user interaction with the physical interface locally managed by the toolkit server; also, returned values from the execution of output events are sent back to the Dialogue Control.

The communication between the Dialogue Control and the toolkit server(-s) is transparent to I-GET programmers. The backbone of such a communication channel is the GTIP (Generic Toolkit Interfacing Protocol). As it has been illustrated in the run-

time architecture of I-GET developed interfaces (Figure 4.2), the Dialogue Control module and the toolkit servers are independent, local or remote, processes. In order to run an interface developed through the I-GET tool, the following default steps are taken (these are managed automatically by the I-GET run-time system and are completely transparent to end-users): (a) an instance (i.e. process) of the Dialogue Control is created locally; and (b) the various toolkit servers, necessary for the running interface, are also locally executed.

There can be many deviations from the default behaviour, which can be defined by the end-user (through special initialization files called "execution plans"). Firstly, the host machines on which the interface client program (the I-GET interface program after being compiled and linked - i.e. the Dialogue Control module in the I-GET run-time architecture), as well as the toolkit servers, will execute, may be explicitly defined. Secondly, not all the servers need to be running. This is particularly required in the following cases: (i) a toolkit server has not been developed yet; or (ii) the software / hardware resources required by a toolkit are not present at the target machine (e.g. audio processing capability, video sampling and presentation, non-visual interaction equipment). The I-GET UIMS supports the development of small and fast toolkit servers, supporting quick response schemes.

In the case of *generalisation*, toolkit interface specification strategies, resulting in *virtual toolkits* within the I-GET language layer, multiple servers must be developed, one for each target toolkit to which the virtual toolkit will be mapped. Interfaces utilising elements of virtual toolkit specifications may run with any of the available servers (but with only one of them for each different execution session); the running interface will have the native "look & feel" of the toolkit associated with the particular running server.

4.3.3.2.1 Generic Toolkit Interfacing Protocol (GTIP)

The Generic Toolkit Interfacing Protocol has been designed to support the asynchronous communication between the Dialogue Control module and toolkit servers. The GTIP reflects the generic toolkit metamodel, which constituted the theoretical basis of the I-GET toolkit interface specification kernel, and is well suited for connecting toolkit-independent software layers with target toolkits. Such a separation is analogous to the case of X WINDOW SYSTEM, where the X server runs locally, handling all screen and device management details, while the client program may run remotely; X client programs are actually linked with the programming interface library for accessing X server facilities. The GTIP is split in two parts: (i) structure of messages that are sent to toolkit servers (DC->TS part, see Figure 4.15), and (ii) structure of messages that are sent from toolkit servers (DC<-TS part, see Figure 4.15). Each message packet has a standard header indicating the type of *request* (DC->TS part), or the type of *notification* (DC<-TS part).

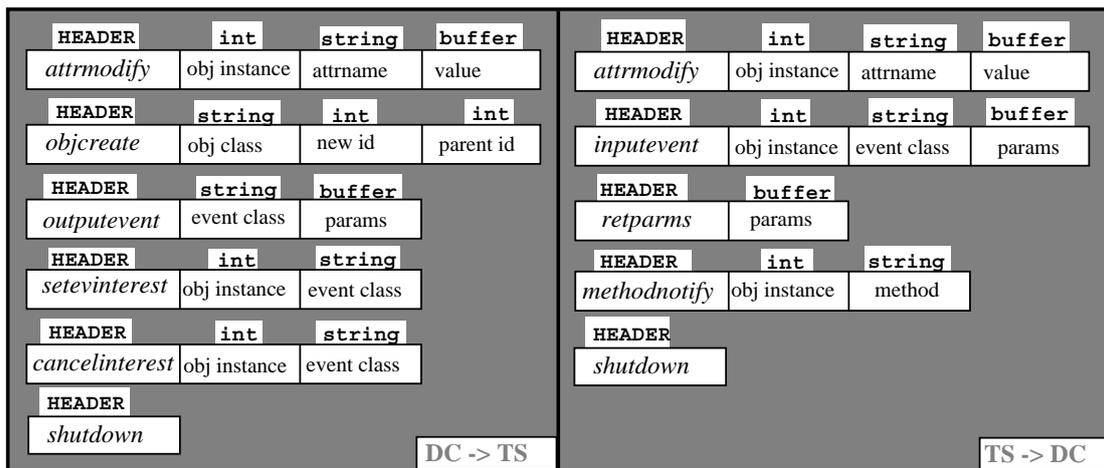


Figure 4.15 - Structure of messages in the Generic Toolkit Interfacing Protocol: from Dialogue Control to Toolkit Server (left part), and from Toolkit Server to Dialogue Control (right part).

It has been tested that the local execution of the toolkit server, in conjunction with the remote execution of the Dialogue Control module, is practically efficient for low-communication lines (e.g. telephone nets), even in cases that simple direct manipulation tasks, like drag-and-drop, are implemented by the client programs. However, the response time has been slower, when a direct manipulation task involved continuous

calls to output event functions having also returned parameters (requiring remote function calls performed by the Dialogue Control, served synchronously by the toolkit server, providing also returned values back to the Dialogue Control).

4.3.3.2.2 Development Complexity of Toolkit Servers for Specific Toolkit Interface Specification Scenarios

There are certain cases in which the introduction of particular programming features, within toolkit interface specification, results into potentially more complex servers. Such inherent complications will mainly require some extra code to be written on the toolkit server side; the most representative of those cases are discussed.

4.3.3.2.2.1 Toolkit Resource Management

In this case, the toolkit interface specification includes output events for manipulating toolkit-specific resources such as: fonts, bitmaps, graphics contexts, etc. When programming with the original toolkit, these resources are managed by toolkit functionality through local specialized types (e.g. **XFontStruct*** in Xlib for fonts, **HBITMAP** for bitmaps in Windows) which are not visible within the I-GET language layer (which supports only the types defined within toolkit interface specification).

<pre>lexical (Xaw); outputevent GetFont (Xaw) [out: string font; in: int fontid;]</pre>	<pre>outputevent TextWidth (Xaw) [out: string text; int fontid; in: int width;]</pre>
--	---

Figure 4.16 - Output events for toolkit resource management.

In order to support such resource management (see Figure 4.16, managing font resources) alternative reference types, mainly integers, are used within the toolkit interface specification (e.g. **int fontid**, see Figure 4.16); the toolkit server should

allocate local resources, generate unique integer identifiers per resource, maintain locally mapping tables between such identifiers and local resources, and return to the Dialogue Control the integer identifier (e.g. `out:` parameter in `GetFont` output event) to be used subsequently by the I-GET programmer as a parameter to the output events for manipulating that particular resource (e.g. `int fontid` in the set of `out:` parameters for `TextWidth` output event).

4.3.3.2.2.2 New Presentation / Behaviour Attributes for Objects

In this case, the specified interaction objects reflect an enriched presentational and / or behavioural structure, with respect to the corresponding original toolkit object classes. The realization of such extra interactive features must be explicitly programmed as part of the toolkit server implementation. For instance, the necessary drawing statements must be programmed for additional presentation attributes (e.g. supporting background bitmaps for object classes which originally do not possess such visual attributes), while code for the extra dialogue facilities emerging from the new behavioural attributes must be appropriately realized (e.g. supporting keyboard manipulation of list boxes for Xaw).

4.3.3.2.2.3 Procedural vs Object Oriented Programming Model Conflict

One typical case in which such conflicts may emerge concerns the introduction of graphic primitives within toolkit interface specification. This type of conflict appears in either of the following cases:

- Having an Object Oriented (OO) toolkit interface specification for a procedural graphics package implementation which is about to be integrated;
- Having a procedural toolkit interface specification for an OO graphics package implementation which is about to be integrated. The toolkit server is required to perform model translation, which introduces considerable complexity; the same issue of model difference has been raised in the context of a GKS library integration

within the SERPENT UIMS, since SERPENT supports only object-oriented entities for the integration process [Bass et al, 1990].

4.3.3.2.2.4 Importing Libraries which Need to be Linked Together

This scenario concerns the situation where various toolkits which have to be integrated, are intended to be utilised together; this is a typical case of unified interfaces, where interaction elements from multiple sources are combined into a single interface implementation, encompassing various diverse dialogue patterns. In order to allow interoperation and data exchange, linking of their respective libraries and object files is required. Moreover, communication and data exchange at the library-call level is needed in certain cases. This scenario cannot be managed via two separate servers, since such library-connections must be established at the same memory space. In this case, a merged server can be built.

4.4 TOOLKIT AUGMENTATION

Augmentation is defined as the design and implementation process through which additional interaction techniques are “injected” within the original interaction elements supplied by a particular toolkit, aiming towards enhancing accessibility and interaction quality for specific user categories. Newly introduced interaction techniques become an integral part of the original interaction elements, while old applications, if re-compiled and / or re-linked with the augmented toolkit version, inherit the extra dialogue features.

Currently, most interactive software products are built by utilising commercially available toolkits, such as the Windows object library. As a result, those software products inherit the advantages, as well as the restrictions, of those interaction elements. For instance, voice control of interaction is not supported in Windows object library; thus, access in a situation where direct visual attention is not possible

(e.g. while driving) cannot be supported. Another typical example where augmentation is required concerns accessibility of windowing interaction by motor-impaired users; we will address in more detail this example in the next Section. In both example cases above, augmentation implies the development of new software interaction techniques, as well as the installation of special purpose I/O devices (e.g. voice I/O hardware, binary switches).

4.4.1 Implementation Requirements

As for toolkit integration, two categories of toolkit augmentation requirements will be defined, each judging the extent to which augmentation is supported by given interface tools: (a) *Minimalistic requirements*, concerning a set of functional capabilities through which augmented object classes can be realistically implemented; the set of these requirements is not tight to any particular programming language, to any type of language (e.g. procedural, object-oriented, scripting); however, it is assumed that the typical development functionality, such as creating object hierarchies, reading or writing object attributes, defining call-backs and implementing event handlers, is supported. (ii) *Maximalistic requirements*, which include the minimalistic requirements, as well as some additional functional criteria, primarily targeted towards enabling an easier and more modular implementation of the augmented object classes.

4.4.1.1 Minimalistic requirements

- Device installation can be supported. This may require low-level software to be written, and it will work either via a *polling-based* scheme (i.e. continuously checking device status), or a *notification-based* scheme (i.e. device-level software may asynchronously sent notifications when device input is detected).
- Manipulation of focus object. During interaction, different interaction objects will normally gain and loose the focus, via user control (e.g. through mouse or tab-keys in Windows) . In order to augment interaction, it is needed to have programming

control of the focus object, since any device input originated from the extra peripherals will always concern the particular object having the dialogue focus.

- Manipulation of the object hierarchy. When implementing augmented interaction, it is necessary to provide augmented analogies of the user-focus control actions; this is required since the user must be enabled to “navigate” within the interface. Hence, the hierarchical structure of the interface objects must be accessed in a programming manner, as part of the navigation dialogue implementation.

The relationship between the native object classes and the augmented object classes is a typical *ISA* relationship, in the sense that augmented classes inherit all the features of the original toolkit object classes (see Figure 4.17). This scheme can be implemented either via sub-classing, if the toolkit is provided in an OOP framework, or via composition, in case of non-OOP classes.

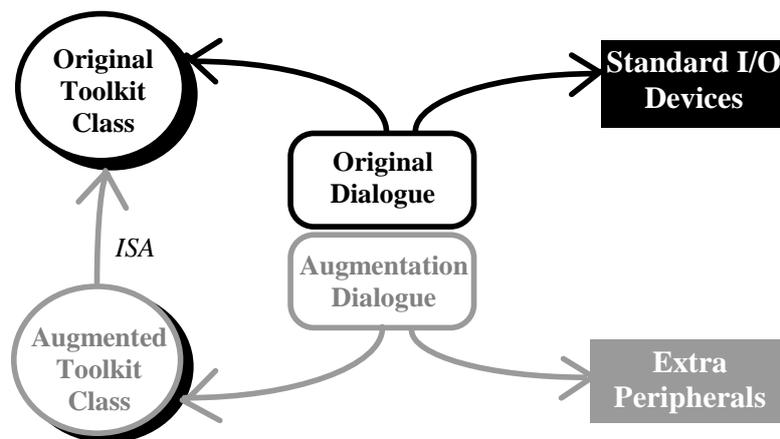


Figure 4.17 - Relationship between original- and augmented- toolkit classes in toolkit augmentation.

The composition is realised by defining an augmented object structure which collects the original as well as the augmented object features, encompassing also an instance of the original object class; this explicit instantiation is necessary to physically realise the toolkit object, which in the case of sub-classing would be automatically carried

out. Also, the interface tool should supply facilities to specify the mapping between the attributes of the toolkit object instance and its corresponding features defined as part of the new object structure (e.g. procedural programming, constraints, monitors).

4.4.1.2 Maximalistic requirements

- Expanding object attributes and methods directly supported by the interface tool on top of the original toolkit classes (see Figure 4.18). This will alleviate the problem of defining new object classes, while it will enable old applications to gain directly the advantages of the augmented dialogue features without modifications (recompilation / relinking merely required).

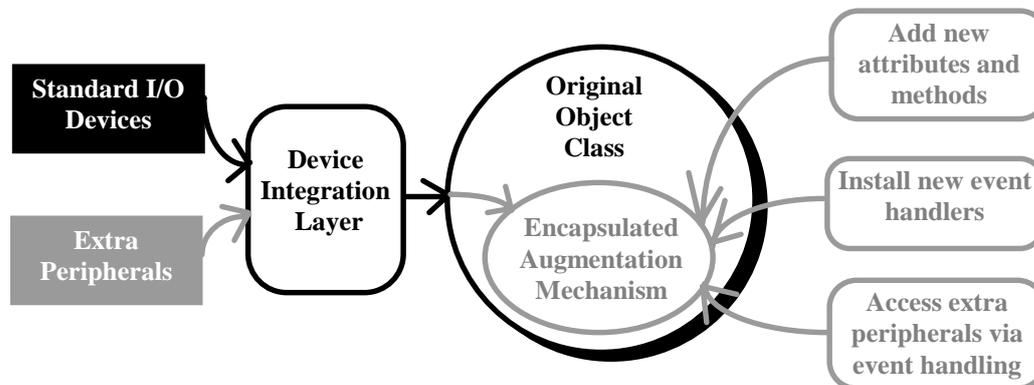


Figure 4.18 - Toolkit augmentation in case that the maximalistic functional requirements are met by a given interface tool.

- Visible and extensible constructor, where additional behaviour can be added. This will allow installation of new event-handlers and the performance of all necessary initialisations directly at the original class level. The constructor may be also supported by non-programming oriented interface tools, by means of user-defined initialisation scripts.
- Modular device installation / integration layer (see Figure 4.18), so that new device input can be attached to toolkit input-event level. This will facilitate the

management of extra peripherals through the original event-management layer of the given interface development tool.

The most important advantage of the maximalistic approach is that it is not required to introduce new specialised object classes, encompassing the augmented interaction features. As a result, it is not inherently necessary to make changes in old applications, as it would be the case in the minimalistic approach, since in the latter, the augmented capabilities are conveyed by newly defined object classes not originally referenced within software applications developed prior to the augmentation process. Naturally, if developers desire to introduce some additional dialogue control logic within old applications, taking advantage of the augmented object attributes and methods, modifications, recompilation and relinking is evidently required in all cases. However, even if no changes are needed in old applications, recompilation or relinking may still be required. For instance, in languages such as C++, if applications are supplied through static linking of toolkit libraries, then relinking is imposed to link with the new augmented library version. If dynamic linking is employed (being the most common case), again relinking is needed to update the run-time reference tables, which are normally included within the application executable, according to the new augmented version of the dynamic library; otherwise, broken references and various run-time errors are likely to occur. In more dynamic languages like Java, this problem is directly solved since every library reference and code dependency is resolved upon execution. As a result, old applications may directly gain the augmented dialogue capabilities upon execution. As it will be discussed later on, the I-GET UIMS supports both the maximalistic requirements, thus enabling augmentation to be effectively carried out, as well as resolves the recompilation and relinking problem in languages such as C++; it will be shown that the former property is due to the capabilities of the I-GET language, while the latter is an advantage of the I-GET run-time architecture.

We will continue by firstly presenting a dialogue design case of augmented interaction facilities, which has been carried out as part of this thesis, which has been also directly exploited, in the context of the ACCESS project [ACCESS Project, 1996]. The designed augmented interaction techniques have been implemented in

C++, resulting in the SCANLIB library [Savidis et al, 1997b], briefly introduced in the Related Work Section. Then, the augmentation support provided by the I-GET UIMS, will be presented showing how maximalistic requirements are practically met.

4.4.2 Designing Augmentation of Windows Object Library with Embedded Scanning Techniques

Currently, existing graphical environments, such as Windows 95™, do not encompass support for access by motor-impaired users which are not capable in using the keyboard and the mouse. This design case has been particularly focused in augmenting Windows interaction controls, in order to enable motor-impaired users, able to only use simple binary switches, to have effective dialogue access. In the design process of the augmented switch-based scanning dialogue methods, we have classified the basic object classes into the following five categories, each requiring a different dialogue policy:

- *Top-level windows*. These are the objects directly providing window management operations. Since, in Windows, it has been impossible to "access" (in a programmable manner) the built-in controls (icons) for window management, it was decided to augment all top-level windows with an additional tool-bar. This tool bar is accessed via the scanning techniques and implements all the window management operations (see Figure 4.19).
- *Container objects*. These are object classes with instances present as intermediate nodes in object hierarchies, able to encompass an arbitrary number of object instances. Container objects enable sequential scanning of all contained objects.

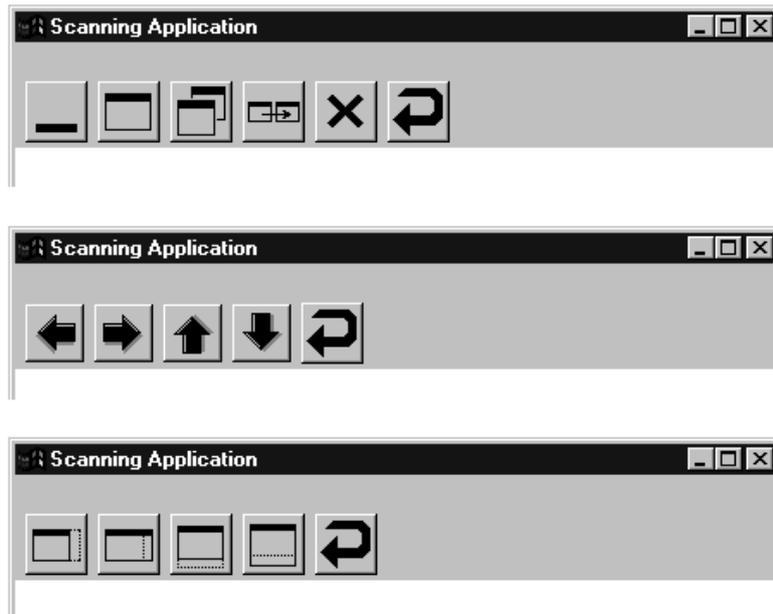


Figure 4.19 - The augmented window management toolbar, presenting three sets of icons (those are display sequentially through the last common icon in each set).

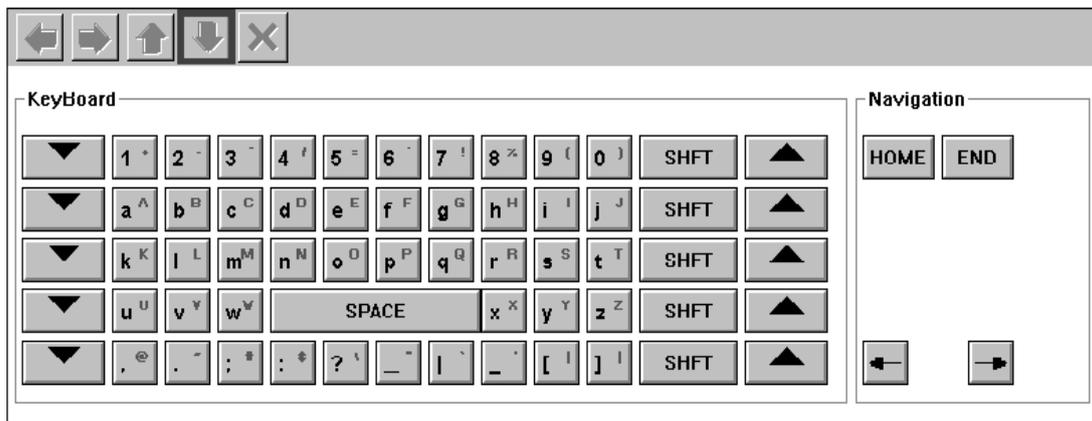


Figure 4.20 - The on-screen pop-up virtual keyboard accessible through scanning interaction techniques.

- *Text-entry objects.* Objects requiring text to be supplied, imposing the need for keyboard emulation. When the user focuses (via scanning) on a particular text-field object, a special keyboard emulation appears automatically, through which, keyboard input is enabled (still via scanning - see Figure 4.20). Such an augmented dialogue is realized transparently to programmers, for which the augmented text-field object at the programming level appears as a conventional Windows text-field object.

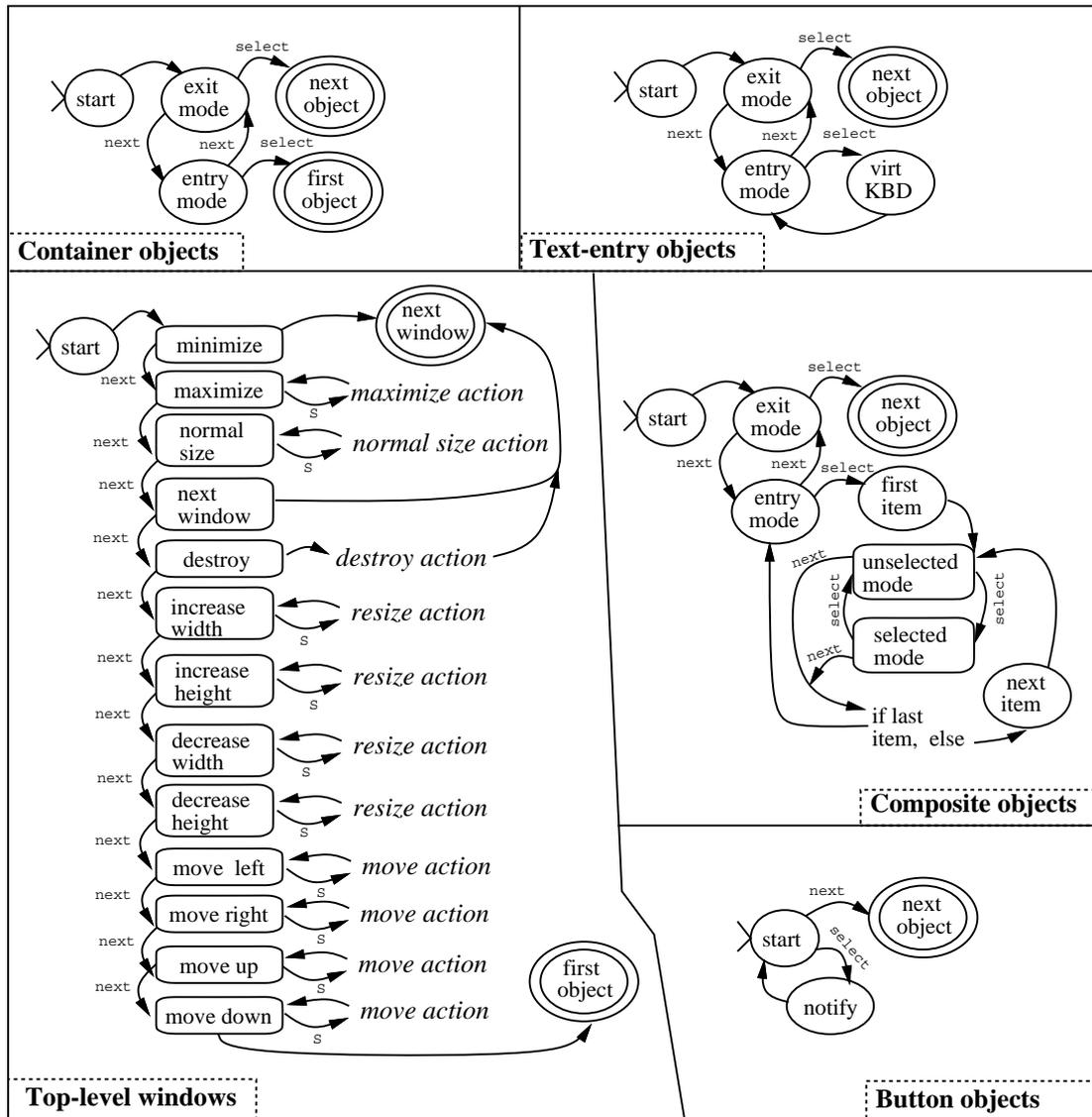


Figure 4.21 - Dialogue structure for the augmented scanning dialogue techniques, based on the fundamental **SELECT** and **NEXT** actions, for the various Windows objects categories defined.

- *Composite objects.* Such object classes provide instances which are leaves in the object hierarchy, however, composed of other objects. Typical examples of composite objects are the scroll-bar class (composed of two arrow buttons and a slider) and the combo-box class (composed of a drop-down button, a label, and a menu object). Composite objects enable sequential scanning of all component-objects.
- *Button categories.* These are simple interaction objects supporting direct user actions for executing associated operations or changing state of boolean parameters. Typical examples from this category are push buttons, check boxes and radio buttons.

The scanning interaction techniques are based on two fundamental actions: *SELECT*, and *NEXT*. The detailed dialogue structure, in a state transition diagram form, is illustrated in Figure 4.21. We provide below the dialogue scenario for container objects, following the designed dialogue structure:

Upon entering a container object, such as a group-box which may enclose an arbitrary number of objects (like other group-boxes, list-boxes, radio-buttons, etc), the dialogue starts in "exit mode". This is indicated to the user via a special highlighting mode. If the user initiates the "select" action (i.e. presses the "select" switch) when in exit mode, the dialogue will move to the next object in the hierarchy, at the same level with the group-box (hence, the user will skip the group-box and all enclosed objects). In case that the user initiates the "next" action, the dialogue is now at "entry" mode (feedback is also provided by changing highlighting style). By initiating a "select" at this state, the user will start dialogue with the first (hierarchically) object of the group-box, while by initiating a "next" action, the dialogue state changes again to "exit" mode (thus, successive "next" actions will circulate between "exit" and "entry" modes). On the basis of these two primitive actions (i.e. "next" and "select"), the dialogue for all the previous categories has been assembled. The following four scanning modes are supported (depending on the type of switch equipment installed):

- 1-switch (for *SELECT* action), time scanning (automatic *NEXT* action, after a specific time interval);
- 2-switches (one for *SELECT* action, one for changing scanning direction), time scanning (as before for automatic *NEXT* action);
- 2-switches (*SELECT* action, *NEXT* action), no time scanning;
- 5-switches (*SELECT* action, *NEXT* action, change direction, exit container, re-start with container).

4.4.3 Toolkit Augmentation Support in the I-GET UIMS

The I-GET UIMS supports two alternative implementation strategies for toolkit augmentation, reflecting two different technical objectives: (i) ensure direct presence of augmented dialogue features in old applications, without imposing recompilation or relinking; or (ii) ensure that maximal capabilities are offered for implementing augmented interaction methods.

On the basis of those two alternative technical goals, the following mechanisms are offered by the I-GET UIMS, respectively: (a) all the various augmented dialogue features are “injected” at the toolkit server side, while I-GET interactive applications, developed prior to the augmentation process, need no extra recompilation / relinking cycles, due to the fact that they never make direct calls to the original toolkit library; or (b) the augmented interaction techniques are implemented through the I-GET language (supporting the maximal requirements), “injected” within the toolkit interface specification layer by ensuring upward compatibility (thus old applications, utilising elements from the old toolkit interface specification version, are not affected). The first method, relying upon the incorporation of augmented dialogue facilities within the toolkit server, is outlined in Figure 4.22. As it is shown, the toolkit interface specification may not be affected, since the extra dialogue techniques which are added at the toolkit server side, do not require any change on the original

toolkit interface specification. Even in cases where the toolkit interface specification has to be expanded, so as to include some newly introduced lexical attributes and / or methods, old applications will still behave correctly without recompilation / relinking. This is due to the fact that the communication among the Dialogue Control and the toolkit servers relies upon the names of elements and their properties; as a result, the original naming conventions for imported elements are fully preserved (only some new attributes and methods may be introduced), and the run-time inter-component communication will still run perfectly, even with the augmented toolkit server version. This technical approach has been taken for the implementation of the augmented interaction techniques for Windows object library (i.e. SCANLIB library [Savidis et al, 1997b]).

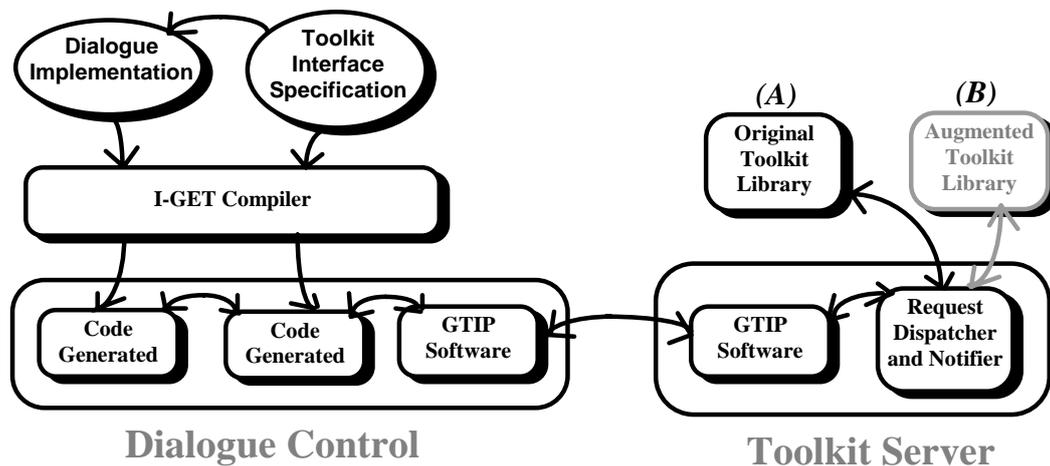


Figure 4.22 - Implementing augmented interaction software at toolkit server side; the Dialogue Control software is not affected, since the transition from the original toolkit library, case (A), to the augmented toolkit library, case (B), takes place at the toolkit server.

In Figure 4.22, the module indicated as “Request Dispatcher and Notifier” has a twofold role: (i) it serves all requests from the Dialogue Control, received via the GTIP layer; and (ii) it sends back notifications, due to user interaction, or returned parameters, due to dispatching of output events.

The second method, in which all augmented dialogue features are implemented through the I-GET language, is based on the extension of the original toolkit interface specification to include the augmented dialogue control logic (see Figure 4.23). This approach is similar to extending C++ classes by adding more member functions, and / or upgrading the implementation of various member functions, without affecting the original class definition (i.e. upward compatibility of class interface); as a result, programs using the old class definition, may be safely recompiled with the new version, gaining directly the upgraded execution behaviour of member functions. In the same manner, I-GET programs (i.e. Dialogue Implementation in Figure 4.23) utilise interaction elements according to their appearance within toolkit interface specification; hence, the programming interface of those elements is not changed, and their respective client dialogue implementations are not inherently affected.

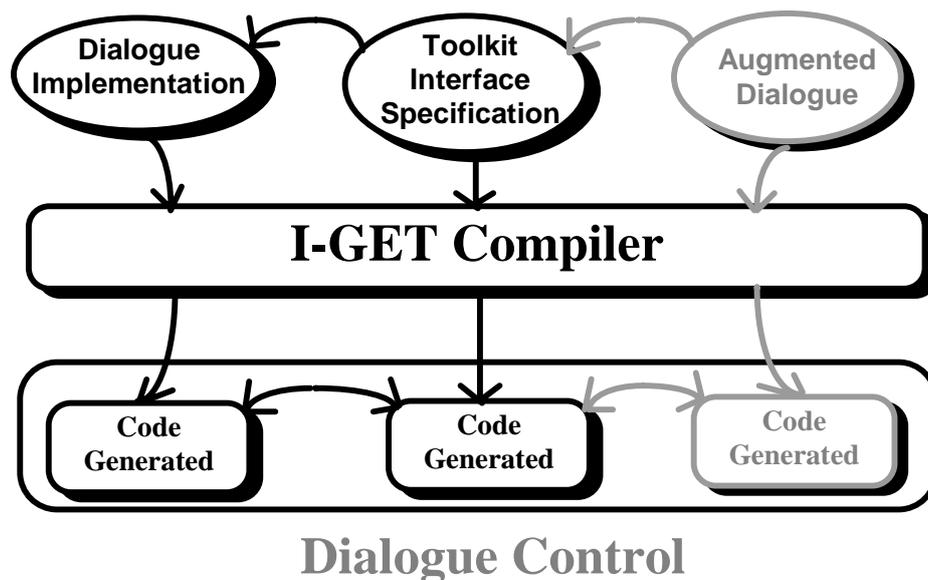


Figure 4.23 - Injecting augmented dialogue code within toolkit interface specification; dialogue implementations utilising the old toolkit interface specification version need only recompilation.

<pre>enum ScanMode=[OneSwTimeScan, TwoSwNoTimeScan, TwoSwTimeScan, FiveSw]; lexical GroupBox (W95) [public: noexport: method GainFocusByScanning; method LooseFocusByScanning; int highlighterBorder=2; string exitModeColor="red"; string entryModeColor="green"; ScanMode scan=OneSwTimeScan;]</pre>	<pre>lexical GroupBox (W95) [public: export: int x, y, width, height; private: noexport: int highlighterWidth; int highlighterHeight; highlighterWidth := width+4; highlighterHeight := height+4; bool entryMode=true; void DrawHighlighter(); void HideHighlighter();]</pre>
<pre>void GiveFocusToNextObject(objectid obj); lexical Button (W95) [method Pressed; eventhandler {me} [(true) SwitchPressed [if (SwitchPressed.which==SELECT) {me}->Pressed; // artificial method notification (AMN) else {me}->LooseFocusByScanning; // AMN]] method {me}.LooseFocusByScanning [// default implementation HideHighlighter(); GiveFocusToNextObject({me});] method {me}.GainFocusByScanning [// default implementation DrawHighlighter();]]</pre>	<pre>void GiveFocusToNextObject (objectid obj) [ObjectClass class; objectid next=FindNextScanningObject(obj, &class); case (class) [Button : [ref lexical(W95) Button button=next; // upcast to object class {button}->GiveFocusByScanning; // AMN to give focus]]]</pre>
<pre>void AddObjectInScanningHierarchy (objectid obj,ObjectClass class); void RemoveObjectFromScanningHierarchy (objectid obj); lexical Button (W95) [constructor [AddObjectInScanningHierarchy({me}, Button);] destructor [RemoveObjectFromScanningHierarchy({me});]]</pre>	<pre>void AddObjectInScanningHierarchy (objectid obj,ObjectClass class); void RemoveObjectFromScanningHierarchy (objectid obj); lexical Button (W95) [constructor [AddObjectInScanningHierarchy({me}, Button);] destructor [RemoveObjectFromScanningHierarchy({me});]]</pre>

Figure 4.24 - Five examples in which the maximal augmentation features of the I-GET language are utilised for implementing scanning-based dialogues.

Next, we will discuss the maximal augmentation capabilities offered by the I-GET language. The implementation of the augmented dialogue implementation can be encapsulated within lexical classes, defined as part of the toolkit interface specification. We will examine the augmentation of Windows object library with embedded scanning techniques as our subject implementation scenario, to demonstrate how such an encapsulation approach is facilitated through specific constructs of the I-GET language.

In Example 1, of Figure 4.24, the introduction of new attributes and methods within the **GroupBox** lexical class is shown, for the **w95** toolkit (i.e. the name with which the Windows 95 is known upon integration in the I-GET UIMS). The original specification of the **GroupBox** class played the role of a class interface specification for toolkit integration purposes; in this sense, it merely included attribute declarations (some with default values as well), and method definitions. In the augmented class definition, the extra attributes and methods reflect the programming control that needs to be offered (to developers) on the new dialogue capabilities, which have been incorporated within the **GroupBox** object class.

The new attributes, **highlighterBorder**, **exitModeColor**, **entryModeColor** and **scan**, are related to the various interaction parameters of the SCANLIB library, as it has been previously discussed. It is important that all newly introduced attributes are completely transparent to the original Windows library; this is due to the fact that the augmentation takes place at the Dialogue Control layer, while toolkit servers should not be aware of such extra attributes. Consequently, there must be a way to specify that all those attributes are hidden from toolkit servers; the **noexport:** qualifier serves this purpose. All attributes declared following a **noexport:** qualification are not communicated to the corresponding toolkit servers; in this context, communication means that an attribute modification statement executed at

the Dialogue Control level at during run-time, is sent to the toolkit server (via GTIP) where it has to be physically executed (i.e. there must be corresponding physical attributes in original toolkit classes) over the real toolkit object instances.

Apart from the **noexport:** qualification, there is a **public:** scope qualifier, meaning that those attributes are visible to the clients of this class (similar to public: semantics in C++). Additionally, two extra method classes, namely **GainFocusByScanning** and **LooseFocusByScannig**, are defined, allowing developers to supply code to be executed when the scanning dialogue focus enters or leaves, respectively, augmented object instances. The set of available methods can be directly expanded for lexical classes, without affecting toolkit servers.

In Example 2, of Figure 4.24, the declaration of private (via **private:**) non-exported objects is shown. Private members support information and implementation hiding, as in typical OOP languages, and are not accessible from outside the lexical class definition body. Three private attributes, namely **highlighterWidth**, **highlighterHeight** and **entryMode**, are defined; the first two are to be used for displaying or hiding the highlighter rectangle by the **DrawHighlighter()** and **HideHighlighter()** functions respectively, while the last is to be used for storing the entry mode (for objects encompassing other objects) during scanning. The two highlighter attributes are defined to be constrained by other attributes of the lexical class: the **highlighterWidth** from **width+4**, and the **highlighterHeight** from **height+4**. Constraints like these, have the following semantics: if any variable engaged in the expression after the **:=** symbol is modified, the expression is evaluated and is assigned on the variable defined on the left of the **:=** constraint symbol.

Constraints are declarative constructs which allow developers to maintain relationships among programming variables easily, with simple formulas, without being concerned with algorithmic matters. Such private local members are important in writing code, encapsulated within lexical classes, for implementing the various presentational and input control aspects of the augmented object dialogue.

In Example 3, of Figure 4.24, the specification of event handlers is mainly illustrated for implementing dialogue management of scanning-based interaction. The event handler is attached to the particular object instance itself (i.e. `{me}` construct), and it shows how the augmented button activation dialogue is implemented. The `SwitchPressed` event causes and event handler to be executed, in which the switch pressed is checked; if the switch has been assigned to the `SELECT` action, the button should be considered to be activated by the user, hence, the `Pressed` method is artificially notified, via the `{me}->Pressed` statement, to cause execution of all user (i.e. developer) callbacks. Else, the switch corresponds to the `NEXT` action, which implies that the scanning focus should be transferred to the next appropriate object. This behaviour is broken in two parts: firstly, the `LooseFocusByScanning` method is artificially notified through the `{me}->LooseFocusByScanning` statement; secondly, in a default, locally defined, implementation of the latter method, the scanning highlighter is hidden, via the `HideHighlighter()` statement, while the scanning focus is given to the appropriate next object, through the `GiveFocusToNextObject()` (this is again a function included within the augmentation code and will be discussed soon). Such support for artificial method notification is crucial for connecting event handling code, which actually manages user input in interaction objects, with methods, providing callback lists for logical actions which are supported by object classes (e.g. selection, pressing, text entry).

In Example 4, of Figure 4.24, the rest of dialogue control for passing scanning focus to the next appropriate object is shown, via the implementation of the `GiveFocusToNextObject()` function. In this function, two are the most important steps: firstly, next object instance, and its respective class, are identified from the stored scanning object hierarchy, via the `FindNextScanningObject()` function; then, from the retrieved generic object reference (i.e. `objectid`), the class-specific reference (i.e. `ref lexical(w95) Button`) is gained, through a case statement, via down-casting, which is used for an artificial notification of the `GainFocusByScanning` method, via the `{button}->GainFocusByScanning` statement. In Example 4, only one code segment is shown, for the `Button` class,

within the case statement; similar blocks are implemented for the rest of augmented object classes.

Finally, within Example 5, of Figure 4.24, the important role of expandable **constructor** and **destructor** blocks is demonstrated. Extra initialisation code is added in the constructor to register the created object instance within the scanning object hierarchy (i.e. `AddObjectInScanningHierarchy()`), while a similar call to cancel the registration made upon instantiation is added within the destructor block (i.e. `RemoveObjectFromScanningHierarchy()`).

The above examples, show how the maximal augmentation features, i.e. expandable attributes and methods, encapsulated event handlers (providing access to standard, as well as to additional input / output devices), and expandable **constructor** / **destructor** blocks, are supported in the I-GET language. These examples also demonstrate how the combination of toolkit interface specification constructs with dialogue implementation facilities and programming-oriented constructs, can enhance the capabilities offered by an interface tool (in the case of the I-GET tool, moving from toolkit integration towards toolkit augmentation support). Such augmentation capabilities, offered by the I-GET language, are not met in other known UIMS tools.

4.5 TOOLKIT EXPANSION

Expansion, over a particular toolkit, is defined as the process through which toolkit users introduce new interaction objects, not originally supported by that toolkit. An important requirement of toolkit expansion is that all newly introduced interaction objects should be made available, in terms of the manipulation facilities offered, in exactly the same manner as with original interaction objects; hence, for toolkit users not aware of the original toolkit classes, it should be indistinguishable which object classes have been implemented as add-ons.

In the context of diverse user requirements and varying situations and contexts of use, it is likely that new interaction paradigms, metaphors and dialogue techniques will be dictated. There are already existing cases clearly demonstrating this need: (a) multimedia CD-ROMs (i.e. information systems), targeted to a broad user population, employ custom-made interaction controls and provide graphical design paradigms clearly beyond the windowing-based traditions; and (b) educational software products, primarily those targeted for young children, employ new metaphors of interaction and practically reject the windowing-based interaction metaphor. However, all these software products still run on top of mainstream graphical toolkits, such as Windows object library. As a result, developers are faced with the problem of expanding the basic interaction facilities with new ones, serving better the needs of their target user groups.

4.5.1 Previous Related Work

Currently, toolkit expansion mechanisms are supplied within various categories of commercial interface tools, while there is a considerable variety with respect to: the way expansion is supported, how difficult it is for developers to define new interaction objects, and the method for employing newly introduced objects within normal interface development. We put particular emphasis on the easiness of the approach, since even though powerful expansion features may be supported, the inherent complexity of the mechanism may pose practical barriers. One of the early toolkits providing expansion support has been the generic Xt toolkit interface, standing on top of the Xlib library for X Windowing System. The Xt mechanism provides a template widget structure where the developer has to provide some implemented constructs. The mechanism of Xt is complicated enough to turn expansion to an expert's programming task. Other approaches to expansion concern toolkit frameworks supported for OOP languages, such as C++ or JAVA. If key super-classes are distinguished, with well documented members, providing the basic

interaction object functionality, then expansion becomes straightforward sub-classing. This is the typical case with OOP toolkits like Windows object library or InterViews. Apart from programming toolkits, the expansion mechanism is also supported in few higher-level development tools, like User Interface Management Systems (UIMS). The demonstration-based method for defining interactive behaviours in Peridot [Myers, 1988], leads to the introduction of interaction objects which can be subsequently recalled and employed in interface construction; hence, in practical terms it can be viewed as an expansion functionality. The Visual Basic tool, for interactive interface construction and scripting, is currently supported with various peripheral tools from third-party vendors; one such tool, called VBXpress, supports the interactive construction of new VBX interaction controls (i.e. expansion).

Finally, the last and more advanced approach to toolkit expansion concerns distributed object technologies for inter-operability and component-based development. New interaction objects can be introduced through the utilisation of a particular tool, while being utilised by another. This functionality is accomplished on the basis of generic protocols for remote access to various software resources, supporting distribution, sharing, functionality exposure, embedding, and dynamic invocation. Microsoft has been the first to allow ActiveX controls to be embedded in JavaBeans containers. This strategy aimed to end-up in a situation where programmers are using JAVA as nothing more than a container to deliver ActiveX functionality on Windows-based computers.

This strategic move has been quickly realised by JavaSoft, which established the ActiveX bridge project, that quickly came-up with Migration Assistant 1.0™, accomplishing exactly the opposite (or symmetric) link: enabling JavaBeans to work inside ActiveX containers. The result, indicated in Figure 4.25, is that today we have software allowing interoperation of ActiveX containers and JavaBeans components in both directions. For programmers of each of those component categories, this capability is an expansion of the set of available interaction controls; for instance, ActiveX programmers are enabled to use directly JavaBeans objects within ActiveX containers.

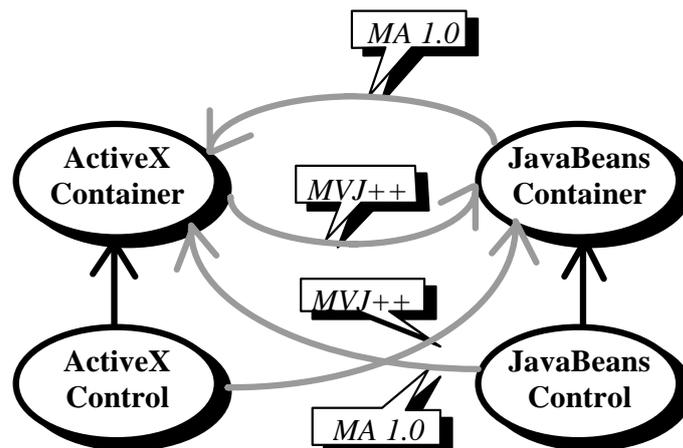


Figure 4.25 - Interoperation among ActiveX and JavaBeans technologies through Migration Assitant 1.0™ (MA 1.0) and Microsoft Visual J++™ (MVJ++).

4.5.2 Implementation Requirements

As it has been the case with the previous two mechanisms discussed, two categories of implementation requirements are defined, each judging the quality and “quantity” of expansion support supplied by a given interface tool: (a) *minimalistic requirements*, which identify the presence of an appropriate object expansion framework; and (b) the *maximalistic requirements*, which judge whether maximal expansion support is practically made available.

4.5.2.1 Minimalistic Requirements

The most typical forms of an expandable object framework, which should be supported by a given interface tool, are listed below:

- Super-class (via derived classes);
- Template structure (by filling implementation gaps);

- API (implementing multiple services complying to a particular API for external clients);
- 4GL object model (interaction objects are defined via dedicated mechanisms in specialised 4GLs);
- Physical pattern (building physically an interaction object around pre-defined physical patterns).

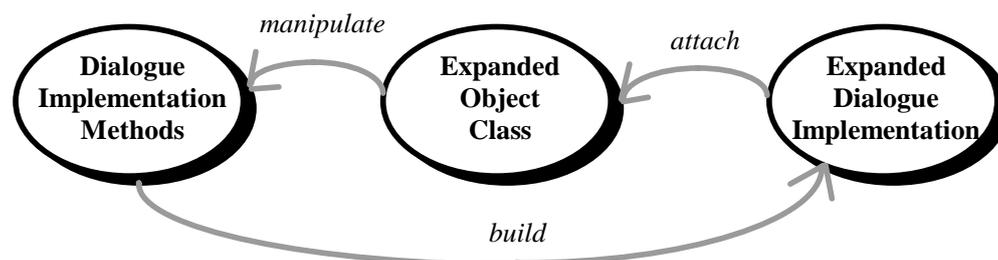


Figure 4.26 - Closure property in maximal expansion - the resulting expanded objects are made through the original dialogue implementation facilities.

4.5.2.2 Maximalistic Requirements

Maximalistic requirements impose only one additional functional property over the minimalistic ones: if an interface tool supports object expansion, then it should be allowed to implement the dialogue for new interaction objects via the native dialogue construction facilities of the interface tool (see Figure 4.26). In other words, developers should be allowed to define dialogues for new interaction objects via the facilities they have been already using for implementing conventional interfaces. For instance, in an interactive construction tool, maximal functionality for expansion is considered to be available, only if interactive object design and implementation is enabled.

4.5.3 Toolkit Expansion Support in the I-GET UIMS

The I-GET UIMS is the first 4GL-based interface tool known to support maximal toolkit expansion. We will present the object expansion framework of the I-GET language on the basis of real development scenarios. More specifically, new interaction object classes have been constructed on top of the Xaw/Athena widget set, working under UNIX OS variants, which has been imported in the I-GET UIMS. Some of those new classes are: a 2D interactive line, a new push-button class, a row / column container, arranging automatically contained objects in a row / column fashion, etc. We will outline the implementation of a new push-button object class, in order to discuss the various expansion language features.

In Figure 4.27, the two major steps in accomplishing toolkit expansion through the I-GET language are shown. The first step concerns the definition of the new lexical class. Such a definition is similar to lexical class definitions met in toolkit interface specification and will provide the programmable picture, from the developers' point of view, of the particular newly introduced object class. The second, and normally the most resource demanding task, is devoted in providing dialogue implementation code; this stage will typically require: (i) the writing code for displaying the object, which may employ graphical primitives as well as other interaction objects (i.e. a composite object class); (ii) implementing event handlers for input management and feedback control; (iii) providing code to map attribute updates on the real object presentational or behavioural aspects (e.g. if the "label" attribute of an object is changed, code must be executed to refresh the displayed object picture according to the new "label"); (iv) notify methods at those points of input management code, where the logical actions represented via methods are considered to be accomplished (i.e. a "radio-button" may fire its "state changed" method upon a mouse click occurring within its screen space).

In Figure 4.28, the **PushButton** expanded object class is outlined; the code segments are split into six logical blocks, each serving a specific distinctive purpose. In block number 1, the class definition header is shown; the class is qualified to be **noexport**, which means that the whole class, including all attributes, is completely hidden from toolkit servers. Hence, instantiations and subsequent attribute modifications on created instances, will never be communicated to the running Xaw/Athena toolkit

server.

1. Class definition	<ul style="list-style-type: none"> • Naming class and associating with proper toolkit name. • Defining attributes and their default values. • Defining method categories.
2. Dialogue Implementation	<ul style="list-style-type: none"> • Implement presentation structure. • Implement input control and feedback. • Map attribute changes to presentation or behaviour. • Notify methods when needed. • Ensure that the object is correctly “hooked” in hierarchies.

Figure 4.27 - The two steps in accomplishing toolkit expansion through the I-GET language.

<code>noexport lexical PushButton (XAW)</code>	1
<code>public:</code>	
<code>int x=0, y=0, width, height;</code>	2
<code>string label="PushButton";</code>	
<code>method Pressed;</code>	
<code>private:</code>	
<code>bool notifyPressed=false;</code>	
<code>notifyPressed: [</code>	
<code> if (notifyPressed==true)</code>	3
<code> {me}-> Pressed;</code>	
<code>]</code>	
<code>daid agentId;</code>	
<code>agent PushButton_Dialogue(</code>	4
<code> int* x, int* y, int* width, int* height, string* label,</code>	
<code> bool* notifyPressed,</code>	
<code> objectid itsParent</code>	
<code>);</code>	5
<code>constructor [</code>	
<code> agentId=create PushButon_Dialogue(</code>	
<code> &x, &y, &width, &height, &label,</code>	
<code> &notifyPressed, {me}.parent</code>	6
<code>);</code>	
<code>]</code>	
<code> destructor [destroy agentId;]</code>	
<code>]</code>	

Figure 4.28 - Outline of the implementation of a new PushButton class for Xaw/Athena widget set via the I-GET language.

The class is called **PushButton** and is associated with the **Xaw** named toolkit. In block number 2, the programming interface of the new class, as seen by clients of this class, is illustrated; the naming and data typing conventions from the Xaw/Athena toolkit interface specification are directly employed. In block number 3, the linkage for method notification is shown. One boolean private attribute, named **notifyPressed** is declared, and a *monitor* is associated to it. Monitors are code segments which are associated to variables, and are executed automatically by the I-GET run-time system when those variables are changed. Hence, in this case, if the **notifyPressed** variable is changed to a true value, a notification for the **Pressed** method, via the **{me}->Pressed** statement, will be generated. As it will be shown, this variable is to be simply set within dialogue implementation, if the **Pressed** method will need to be notified.

In block number 4, the dialogue control component header for managing all presentation and input control details is provided; the component is named **PushButton_Dialogue**, while it is defined via the **agent** keyword. Dialogue control components in the I-GET language are called *agents*; agents can be parameterised, and may encompass object instances, event handlers, functions, method implementations and various other categories of constructs. The details of agents will be discussed later on, showing how style manipulation and coordination is facilitated for assembling unified interface implementations.

In block number 5, an agent instance is created within the constructor block, via a **create** statement. Such an instantiation will, among others, cause: activation of all embedded event handlers (defined as part of the respective agent class), registration of all local callbacks, and instantiation of all local (within the agent class) member objects. As it is shown, all object class attributes are passed as pointer parameters to such an agent class; the reason is that from within the agent class, all the attributes of the **PushButton** class need to be directly accessed (which is enabled via pointers). Similarly, a pointer to the **notifyPressed** variable is passed; through that pointer, the content of the **notifyPressed** variable will have to be set, within the event

handlers of the `PushButton_Dialogue` agent, during input control management, to cause (via its associated monitor) method notification. The I-GET run-time system is capable of detecting even such indirect changes (i.e. via pointer references). One of the parameters to the dialogue control agent class is the parent object of the particular `PushButton` object instance (i.e. `{me}.parent`). The latter is necessary, since, within the agent class, a `Simple` widget instance (i.e. something like a canvas object) will have to be declared, providing the screen space for the `PushButton` class; that instance will have to accept as parent object the specific parent object of the `PushButton` instance.

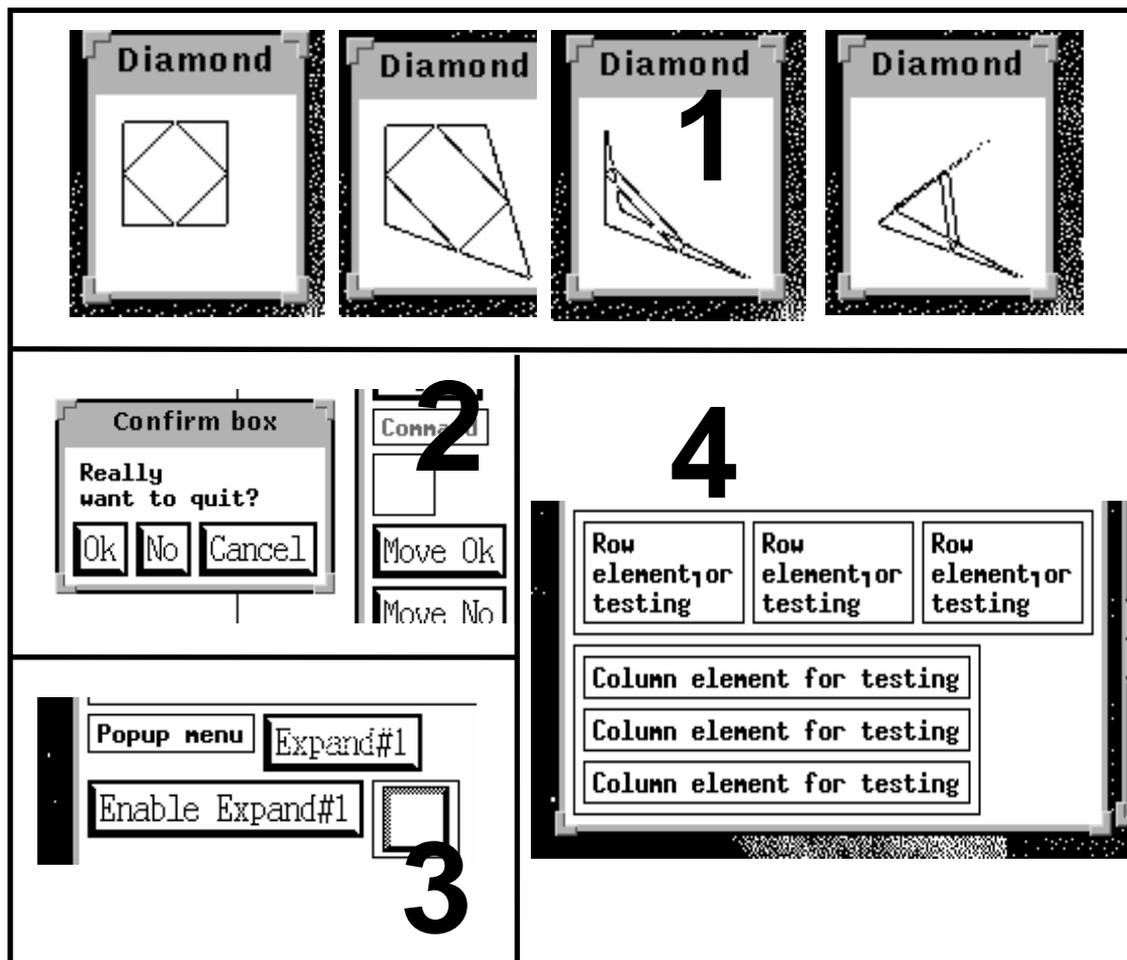


Figure 4.29 - Screen snapshots from expanded object instances, on the Xaw/Athena widget class, implemented via the I-GET UIMS: (1) interactive 2D lines, in which constraints have been applied to force connection among starting- and ending- points;

(2) push-buttons, used in a confirm box; (3) enabling / disabling push-buttons (greyed presentation in disabled mode); and (4) row / column containers.

Finally, in block number 6, the `agentId` variable, which has been used to store the agent instance identifier created upon object instantiation, is now passed to a `destroy` statement, to delete that agent instance.

In Figure 4.29, the expanded object classes for Xaw/Athena widget set, which have been developed through the I-GET language, are illustrated within snapshots from I-GET developed applications.

4.6 TOOLKIT ABSTRACTION

Toolkit abstraction is defined as the ability of the interface tool to support manipulation of interaction objects which are thoroughly relieved from physical interaction properties. Abstract interaction objects are high-level interactive entities reflecting generic behavioural properties, having no input syntax, interaction dialogue, and physical structure. An example of an abstract interaction object is provided in Figure 4.30, where an abstract *selector* object is illustrated. Such an abstract interaction object has only two properties: the number of options and the selection (as a number) made by the user.

Additionally, it may encompass various other programming attributes, such as a call-back list (i.e. reflecting the *select* method), and a boolean variable to distinguish among multiple-choice and single-choice logical behaviours. As is illustrated within Figure 4.30, multiple alternative physical styles, possibly corresponding to different interaction metaphors, may be defined as physical instantiations of the abstract *selector* object class. When designing and implementing interfaces for diverse user groups, even though considerable structural and behavioural differences are naturally

expected, it is still possible to capture various syntactic commonalities, by analysing the structure of sub-dialogues at various levels of the task hierarchy.

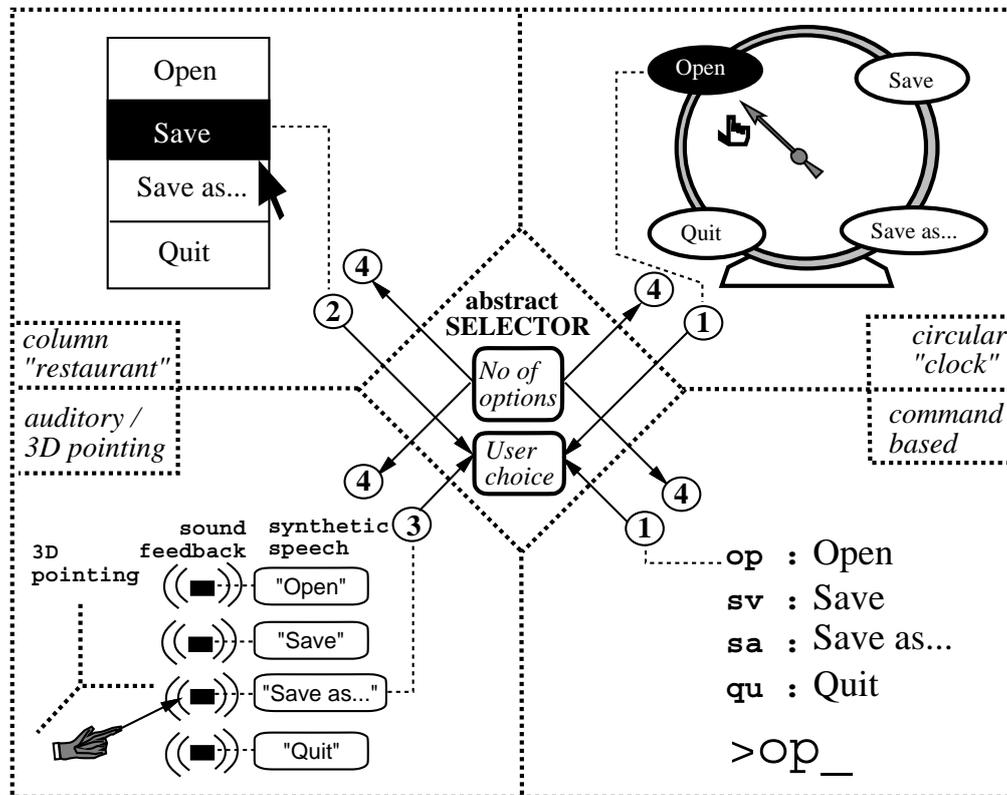


Figure 4.30 - An abstract SELECTOR interaction object class, having only two abstract attributes (central diamond); four alternative physical realisations are shown.

In order to promote effective and efficient design, implementation, and refinement cycles, it is crucial to express such shared patterns into various levels of abstractions, in order to support modification only at a single abstract level, by automating the propagation to the various alternative physical artefacts. Abstract interaction objects can be employed for designing and implementing dialogue patterns which need to have no physical interaction properties. In this sense, such dialogue patterns are not restricted to any particular user group or style of interaction. The introduction of the intermediate physical instantiation levels, so that abstract forms can be mapped to concrete physical structures is also required. By automating such an instantiation mechanism, development for diverse users is facilitated in a unified fashion (at an

abstract layer), while the physical realisation is automated on the basis of an appropriate object instantiation mechanism.

4.6.1 Previous Related Work

4.6.1.1 Generalising and Reusing Primitive Input Behaviours

The key discipline in such research efforts is that various input behavioural aspects of interaction in direct manipulation graphical interfaces are captured, generalized and expressed via a set of generic highly-parameterized primitives through which complex behaviours can be assembled in a modular manner. Initial work in this field is mainly concerned with *interaction tasks* [Foley et al, 1984], providing a comprehensive theoretical framework, while *interactors* [Myers, 1990] provided an implementation-based framework for combining input behaviours to create complex interactions. Both addressed the desk-top metaphor for graphical interaction and propose an alternative structural and behavioural "view" of the lexical level; hence, they are not related to the abstraction domain.

4.6.1.2 Visual Construction with Behaviour Abstraction

This approach has been realized initially in the ADG system (Application Display Generator) [Desoi et al, 1989] and is based on the visual construction of interfaces on the basis of behavioural abstraction of interaction objects; the developer may choose among different physical realizations of such abstract behaviours. Through such different choices for physical realization of abstract objects, alternative versions of the lexical structure of the developed interface are produced. The abstractions supported in the ADG system are visual objects like "bar gauges", "scrollbars", etc, which can have alternative representations. The number of such abstract objects, as well as the links between abstract objects and specific physical alternatives are fixed in the ADG system. The previous type of "abstraction" should be better called *generalization*, since the resulting objects classes are still related to the desk-top metaphor, merely providing a collection of generically applicable desk-top behaviours.

4.6.1.3 Formal Models of Interaction Objects

Such models have been strongly related to methods for automatic verification of interactive systems, in which formal specification of the User Interface is employed to assert certain properties; the theoretical background of such approaches is reactive systems from distributed computation theory. The most representative example is the Interactor model [Duke et al, 1993], [Duke et al, 1994]; interactors convey both state and behaviour, communicate with each other, as well as with the user and the underlying application. Such an entity category is computationally very primitive (as a result, highly generic), and is appropriate for modeling in detail the dialogue of objects from within (i.e. the physical dialogue of an object as such). The interactor model itself constitutes an algorithmic computation structure, like state automata, and as a result, is far away from typical programming models of interactive entities like interaction objects in toolkits.

Hence, interactors may be highly expressive to allow interaction objects to be formally represented as collections of communicating interactor instances; however, they are less convenient as a development entity in an interface tool (in the same manner that the Turing Machine, though the most generic algorithmic structure, is not employed as an explicit programming model in programming languages). In conclusion, the interactor model, though being highly generic, is better suited for formal verification approaches (and has been very successful in that respect), rather than for interface development languages.

4.6.1.4 Meta-widgets and Virtual Interaction Objects

The concept of meta-widgets is based on the abstraction of interaction objects above physical platforms [Blattner et al, 1992]. A meta-widget is free of physical attributes and presentation issues and is potentially capable to model interaction objects above metaphors. Currently, meta-widgets have been provided with a fixed implementation [Wise et al, 1995], with respect to the classes of meta-widgets and their physical instantiation. The notion of *virtual objects*, originally referring to multi-platform

objects [Myers, 1995], has been revisited and enhanced in [Savidis et al, 1995a] to indicate abstraction on top of toolkits and interaction metaphors; implementation support for virtual objects has been provided in the context of the HOMER UIMS [Savidis et al, 1998], in which dual physical instantiations for virtual objects have been supported. The notions of meta-widgets and virtual objects are similar at the high-level, though virtual objects have been accompanied, in the context of the HOMER UIMS, with more concrete engineering models.

4.6.1.5 Overall Conclusions from Related Work

In conclusion, it was found that past work has not addressed the provision of interface development techniques for: (i) creating abstract interaction objects; (ii) defining alternative schemes of mapping abstractions to physical entities (i.e. polymorphism allowed); and (iii) selecting the desirable active mapping schemes for abstract object instances, in the context of interface implementation. Additionally, the ability to handle multiple toolkits, as it is fundamentally required for building unified interface implementations, is not supported. The absence of such development methods from existing tools is due to the different development needs of typical graphical interactive applications, with respect to unified interfaces. All the above technical issues become top priority in engineering unified interactions.

4.6.2 Implementation Requirements

The notion of abstraction has gained increasing interest in software engineering as a solution towards recurring development problems. The basic idea has been the establishment of software frameworks clearly separating those implementation layers relevant only to the nature of the problem, from the engineering issues which emerge when the problem class is instantiated, in various different forms. The same philosophy applies to the development of interactive systems, where abstractions for building dialogues are employed, so that a dialogue structure composed of abstract objects can be re-targeted to various alternative physical forms, by means of an

automation process controlled by the developer. Here, we will firstly try to draw the complete portrait of what abstraction means in the context of interaction objects, comparing also with previous work and definitions, where the concept of abstraction has been explicitly engaged. Then, we will identify the most important functional needs that interface tools should meet, so that manipulation of object abstractions can be supported.

Two categories of functional requirements are distinguished: (a) Minimalistic requirements, which provide a basic set of functional criteria judging if interface development tools facilitate development based on abstract objects; additionally, we also discuss some high-level implementation issues, revealing the complexity of explicitly programming abstract objects, if the interface development tool does not support them explicitly. (b) Maximalistic requirements, which judge if an interface tool provides powerful methods for manipulating abstractions, such as: defining, instantiating, polymorphosing, and extending abstract interaction object classes.

4.6.2.1 Decoupling Abstract Syntactic Behaviour and Morphology

Even though the interactive behaviour of interaction objects is characterized by certain properties such as "look and feel", there are many situations in which differentiated physical realizations reflect similar or even identical behaviours. The level of differentiation varies from simple presentational differences (e. g. consider realization of a "menu" behaviour for different toolkits such as X/Athena, OSF/Motif, X/InterViews, Windows 95), to radically different morphological structures and interaction policies (as shown in Figure 4.30). All four physical interaction objects of Figure 4.30 allow some kind of selection from an explicit set of options (i.e. "menu" behaviour, i.e. abstract selector, as illustrated in the diamond in the centre). Consequently, it is possible to decouple the higher-level syntactic behaviour from the specific morpholexical realization (i.e. "look & feel"), through which the behaviour is physically perceived and understood by the user.

Behaviour abstraction techniques, as they have been applied in systems like ADG [Desoi et al, 1989], and Jade [Zanden et al, 1990], mainly relate to toolkit retargetability [Cowan et al, 1993] (i.e. running the same interactive application over different target toolkits). Those schemes of behaviour abstraction are still within the bounds of the desk-top metaphor (i.e. they introduce visual attributes, typically met in interaction objects of windowing based environments), while the supplied set of abstract behaviours is fixed (provided as a non-expandable library to interface designers). The concept of meta-widgets [Blattner et al, 1992] is more close to the notion of abstract interaction objects; however, the current realization of meta-widgets is restricted to a fixed implementation framework [Wise et al, 1995], composed of hard-coded programming classes, regarding both the basic meta-widgets categories, as well as their respective physical implementations.

4.6.2.2 Minimalistic Requirements

- The interface tool supplies a pre-defined collection of abstract interaction object classes - *closed set of abstractions*;
- For each abstract object class, there is a pre-defined mapping scheme to various alternative physical object classes - *bounded polymorphism*;
- For each abstract object instance, the developer may select which of the alternative physical classes (in the mapping scheme) will be instantiated - *controllable instantiation*;
- More than a single physical instance may need to be active for a particular abstract object instance - *plural instantiation*.

The above requirements imply that the developer is enabled to instantiate abstract objects, while having control on the physical mapping schemes that will be active for each abstract object instance; mapping schemes define the candidate physical classes

for physically realising an abstract object class. The need for having multiple physical instances active, all attached to the same abstract object instance (i.e. plural instantiation), is imposed in case that a running interactive application maps to multiple concurrent physical interfaces. This is a typical case for applications in the field of Computer Supported Cooperative Work (CSCW), and it is also needed in the context of Dual interface development [Savidis et al, 1995a], where two concurrently active instances are required (i.e. a visual and a non-visual) for each abstract object instance. The notion of polymorphic physical mapping, as well as plural instantiation, have fundamentally different functional requirements, with respect to polymorphism of super-classes in OOP languages. The key differences are outlined in Figure 4.31.

Polymorphism in unified development	Polymorphism in OOP languages
<ul style="list-style-type: none"> • Aims to support various dialogue faces (i.e. polymorphism with its direct physical meaning). 	<ul style="list-style-type: none"> • Aims to primarily support re-use and implementation independence (i.e. polymorphism with its metaphoric meaning).
<ul style="list-style-type: none"> • Instantiation must be applied on abstract classes. 	<ul style="list-style-type: none"> • Instantiation is always applied on derived non-abstract classes.
<ul style="list-style-type: none"> • Multiple physical instances, manipulated via the same abstract object instance, may be active in parallel. 	<ul style="list-style-type: none"> • References to an abstract class may be mapped only to a single derived object instance.

Figure 4.31 - Key differences, with respect to polymorphism functional requirements, between OOP development languages and unified interface support.

Clearly, the traditional schema of abstract / physical class separation in OOP languages by means of class hierarchies and *ISA* relationships cannot be directly applied for implementing the abstract / physical class schema as needed in interface development. An explicit run-time architecture is required, where connections among abstract and physical instances are explicit programming references, beyond the typical *instance-of* run-time links from *ISA* hierarchies.

4.6.2.3 Maximalistic Requirements

Apart from controllable instantiation and plural instantiation, as they have been defined in the context of the minimalistic requirements, the following are also required:

- Facilities to define new abstract interaction object classes - *open abstraction set*;
- Methods to define alternative schemes for mapping abstract object classes to physical object classes; e.g. an abstract “selector” may be mapped to a visual “column menu” and a Rooms non-visual “list-box” - *open polymorphism*;
- Mechanisms for defining run-time relationships between an abstract instance and its various concurrent physical instances; this may require the definition of attribute dependencies and propagation of call-back notifications (i.e. if a particular physical instance is manipulated by the user, the abstract instance must be appropriately notified) - *physical mapping logic*;
- Enabling direct programming access, through the abstract object instance, to all associated (concurrently) physical instances - *physical instance resolution*.

4.6.3 Toolkit Abstraction Support in the I-GET UIMS

The I-GET language provides implementation constructs which fulfil the maximal requirements of toolkit abstraction. Abstract interaction objects, which are called virtual objects in the I-GET language, may be defined through a mechanism called *virtual object genesis*. Virtual objects exhibit no physical interaction properties by themselves, unless being associated with a *lexical instantiation relationship*. Such relationships may encompass multiple alternative *instantiation schemes*, each defining a particular mapping of a virtual objects class to a specific *lexical object class*. Since in the I-GET language multiple named toolkits, called *lexical layers*, may

be managed in parallel, it is allowed to define one instantiation relationship, for a given virtual object class, for each such imported toolkit. This situation is illustrated in Figure 4.32, showing all the intermediate links for bridging virtual interaction objects to lexical interaction objects; the latter, may be either implemented directly by the underlying toolkit, or may result from the application of the toolkit- augmentation, or expansion mechanisms. Also, within Figure 4.33, the various implementation levels for interaction objects in the I-GET language are shown, reflecting the various links shown in Figure 4.32.

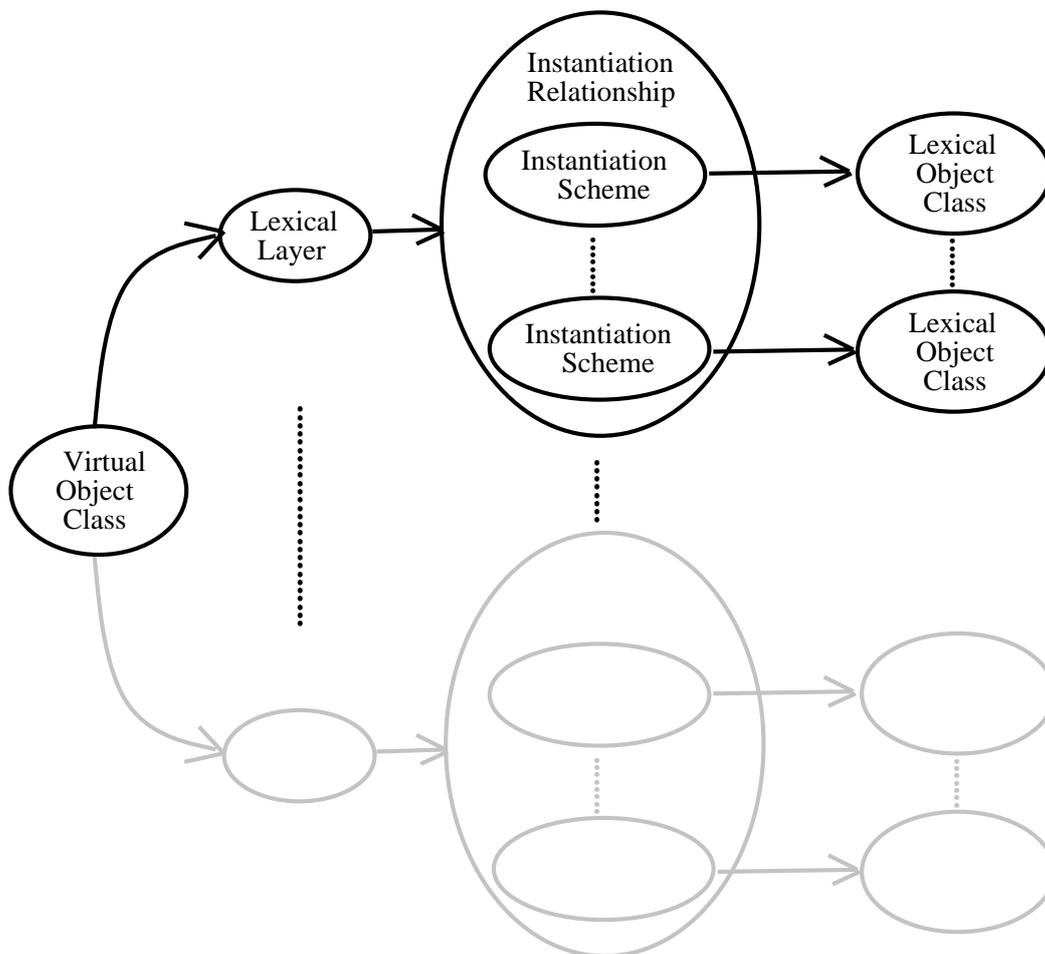


Figure 4.32 - The links between virtual object classes, lexical layers, instantiation relationships, instantiation schemes and lexical object classes, in the I-GET language.

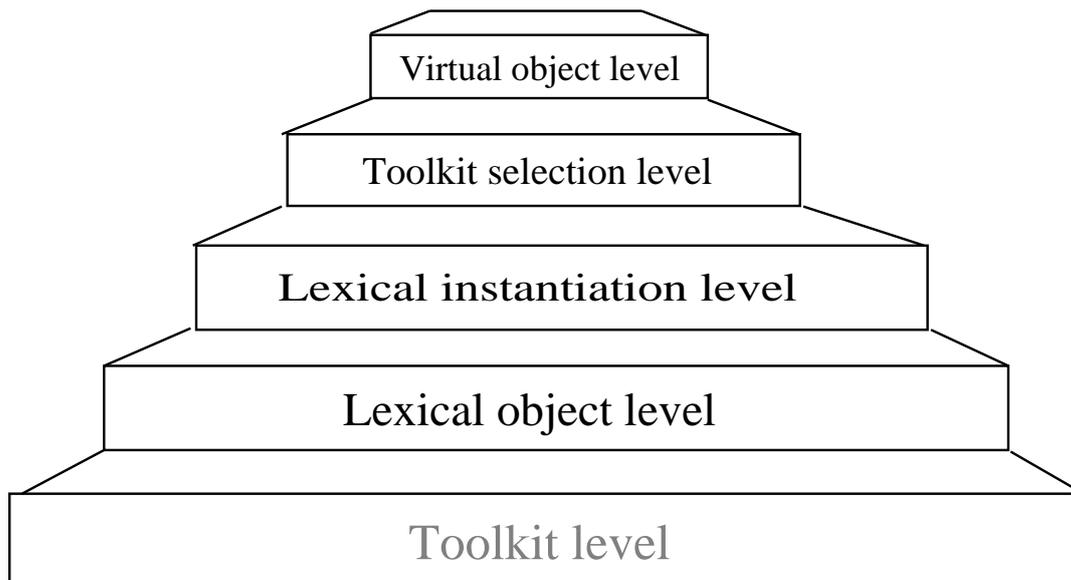


Figure 4.33 - Implementation layers for interaction objects in the I-GET language.

In order to discuss the maximal abstraction capabilities of the I-GET language, we will use some code excerpts taken from the ACCESS project [ACCESS Project, 1996], where the I-GET UIMS has been used for defining and using abstract objects [ACCESS D142, 1997], [ACCESS D132, 1995].

This discussion will flow in the following manner: firstly, we will define a virtual “push-button” class, exhibiting no physical interaction properties; then, we will define two instantiation relationships, one for the Windows object library toolkit and one for the HAWK toolkit [Savidis et al, 1997c], both of which have been integrated within the I-GET UIMS [ACCESS D151, 1995], in the context of the ACCESS project; following the full definition of the physical instantiation logic of the “push-button” object, we will show the declaration of virtual instances, programming access on virtual attributes and implementation of virtual methods, as well as programming access on lexical attributes and implementation of methods for lexical instances.

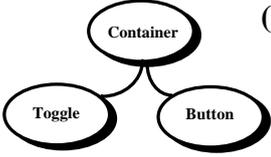
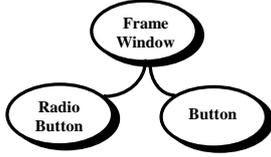
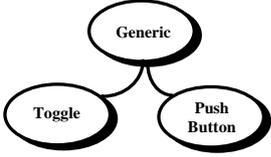
<pre> virtual Button (Xaw,W95,Hawk) [bool Accessible=true; method Pressed; constructor [] destructor []] </pre>	1	<pre> virtual Selector (Xaw,W95,Hawk) [bool Accessible=true; method Selected; int NumOfOptions=0; int UserChoice=-1; constructor [] destructor []] </pre>
<pre> instantiation Button (W95) [// instantiation relationship for Windows Button : Button [{me}Accessible := {me}W95.Accessible; method {me}W95.Pressed [{me}->Pressed;] constructor [] destructor []] default Button;] </pre>	2	
<pre> instantiation Button (Hawk) [// instantiation relationship for Hawk PushButton : PushButton [{me}Accessible := {me}Hawk.accessible; method {me}Hawk.Activated [{me}->Pressed;] constructor [] destructor []] default PushButton;] </pre>	3	
<pre> virtual Container cont : scheme(W95)=FrameWindow; virtual Button button : parent(W95)={cont}W95 : parent(Hawk)={cont}Hawk; virtual Toggle toggle : scheme(W95)=RadioButton : parent(W95)={cont}W95 : parent(Hawk)={cont}Hawk; </pre>	4	
<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;">  <p>(A)</p> </div> <div style="text-align: center;">  <p>(B)</p> </div> <div style="text-align: center;">  <p>(C)</p> </div> </div>		
<pre> Color color=[0,255,0]; {button}W95.bkColor=color; {button}Hawk.soundFile="press.wav"; {button}Hawk.label={button}W95.label="Press"; {button}.Accessible=false; </pre>	5	
<pre> method {button}.Pressed [printstr("Pressed!")] method {button}Hawk.Activated [printstr("Blind user did it!");] method {button}W95.Pressed [printstr("Sighted user did it!");] </pre>	6	

Figure 4.34 - (1) Virtual class definitions; (2) instantiation relationship for W95; (3) instantiation relationship for Hawk; (4) A, B and C are resulting virtual-, W95- and

Hawk- instances hierarchies, respectively; (5) accessing virtual and lexical attributes; and (6) implementing virtual and lexical methods.

In Example 1 (left), of Figure 4.34, the specification of the **Button** virtual class is defined. The class definition starts with the **virtual** keyword, followed by the class name and the explicit list of toolkit names, for which the virtual class will be allowed to be given instantiation relationships, being in our example **Xaw**, **W95** and **Hawk** (the order of appearance is not important). Hence, practically speaking, the purpose of the list is to define for which toolkits the newly introduced class constitutes an abstraction. Within the virtual class, one virtual attribute, called **Accessible** (to enable / disable dialogue with the user), and one virtual method, called **Pressed**, are locally defined. It is evident that such an object class forms a pure abstraction for all physical interactive behaviours, independently of particular style and / or metaphor, supporting direct user actions (like a button-press). Even though its particular name, i.e. **Button**, is related to a specific real-world metaphor, i.e. electric devices, in the I-GET language context it merely has a notational value (i.e. it is just a class identifier); a better name may be devised, more indicative of its generic role.

Similarly, the **selector** class, encompasses only those attributes which are relieved from physical interaction properties, and a single abstract method called **Selected**. Apart from the **Accessible** attribute (which can be seen as a standard property for all virtual classes), it has two additional attributes: **NumOfOptions**, indicating the total number of items from which the user may select, and **UserChoice**, which contains the order of the option most recently selected by the user. It might be noticed that the list of options is not stored within the virtual object class; this has been a careful design decision, since options do not always have a specific information type, but may vary for different physical interaction styles. For instance, options may have: textual, iconic, animated, video, or audio content. Hence, the storage of options is moved to be part of the physical interaction object classes. For all virtual classes, a constructor / destructor block is supported, for code that needs to be executed upon object instantiation / destruction.

Within Examples 2 and 3, of Figure 4.34, the specification of instantiation relationships for the **Button** virtual class, for the **w95** and **Hawk** toolkits, respectively, is provided. An instantiation relationship encompasses the implementation logic for: (i) mapping a virtual object class to a lexical object class; and (ii) defining the run-time relationship among virtual and lexical attributes / methods. An instantiation relationship may encompass an arbitrary number of instantiation schemes, which are defined as follows:

1. An arbitrary name is given to each instantiation scheme, which has to be unique across its *owner* instantiation relationship block (i.e. defined in-between an opening [and a closing square bracket]). For instance, the name **Button** is given, to the scheme defined within Example 2, while the name **PushButton** is similarly given to the only scheme defined as part of the **Hawk** instantiation relationship of Example 3.
2. Each scheme maps the virtual object class to a particular lexical class. The name of this lexical class must be provided next to a colon symbol **:**, directly following the scheme name. For example, the **Button** lexical class for **w95** toolkit is chosen as the target lexical class of the **Button** scheme in Example 2, while the **PushButton** class is similarly defined for the **PushButton** scheme in Example 3. We have employed the convention of naming schemes with the names of their respective lexical classes.
3. In the body of instantiation schemes, the linkage between the virtual and lexical instances is implemented. Normally, constraints and monitors will be employed for bridging together virtual and lexical attributes, while artificial method notifications are employed to fire virtual methods, due to lexical method activation. For instance, in Example 2, the lexical attribute **Accessible**, of the **w95 Button** class, syntactically defined as **{me}w95.Accessible**, is constrained by the **Accessible** virtual attribute, syntactically defined as **{me}.Accessible**. Also, a local implementation of the lexical **Pressed** method is supplied, which executes only a single statement, generating a notification of the virtual **Pressed** method; this

establishes the link for propagating method notifications from the lexical object instances, up to virtual object instances.

4. The default active instantiation scheme is chosen; in Example 2, the expression **default Button**; defines that **Button** is to be activated as the default lexical instantiation of declared **Button** virtual object instances, for the **w95** toolkit (unless another existing scheme is explicitly chosen, as it will be explained later on).

At run-time, when an instance of a virtual class is declared, the various instantiation relationships, for each toolkit listed in its class definition header, are firstly identified. Then, for each instantiation relationship, the necessary instantiation scheme is activated; this may be the default scheme, or a scheme explicitly chosen by the interface developer (the details of declaring virtual object instances will be discussed next). The scheme activation step results in making an instance of the lexical class which has been associated to that scheme; this lexical instance is attached to the original virtual instance. Finally, the various constructs defined locally, within the scheme body are activated, establishing the run-time links among the virtual and the physical object instances.

In Example 4, of Figure 4.34, the declaration of various virtual object instances is illustrated. The three most important properties of virtual instance declarations are: (a) physical instantiations will be made automatically for each target toolkit (i.e. *plural instantiation*); (b) explicit scheme selection is allowed for each toolkit, hence, enabling the developer to choose the desirable physical instantiation of a virtual instance for each toolkit (i.e. *controllable instantiation*); and (c) an appropriate parent lexical instance should be explicitly supplied, for each toolkit defined (i.e. *polymorphic object hierarchies*). In Example 4, the expression **:scheme(W95)=FrameWindow** makes an explicit scheme selection for the virtual instance called **cont**. Similarly, the **:parent(Hawk)={cont}Hawk** expression, for the **button** virtual instance, means that its parent instance for the **Hawk** toolkit is the **Hawk** lexical instance, retrieved from the **cont** virtual instance; in pictures (A), (B) and (C) of Figure 4.34, the virtual-, **w95**- and **Hawk**- instance hierarchies are

indicated, all resulting from the declarations of Example 4. In Example 5, access on virtual and lexical attributes is demonstrated. For a virtual instance $\langle obj \rangle$, the conventions $\{ \langle obj \rangle \}. \langle attr \rangle$ and $\{ \langle obj \rangle \} \langle toolkit \rangle. \langle attr \rangle$ represent a virtual attribute $\langle attr \rangle$ and a lexical attribute $\langle attr \rangle$, the latter to be found in the active lexical instance of $\langle toolkit \rangle$ toolkit, respectively. Hence, following Example 5, for the **button** virtual instance, the $\{ \text{button} \} \text{w95}. \text{bkColour}$ is an attribute of the **w95** active lexical instance, which, due to the fact that the active **w95** scheme for button is **Button**, must be an attribute of the **Button** lexical class.

Finally, within Example 6, the implementation of callbacks for either the virtual or lexical instances is shown. The $\{ \text{me} \}. \text{Pressed}$ method is a virtual method, which, according to the specifications in Examples 2 and 3, will be fired if either the **Hawk** or the **w95** instances are notified; hence, a *unified method implementation* is allowed. If, however, further specialisation of method behaviour is needed for each toolkit, the callbacks of the appropriate lexical instances must be accessed; as it is shown in Example 6, the $\{ \text{me} \} \text{Hawk}. \text{Activated}$ and the $\{ \text{me} \} \text{w95}. \text{Pressed}$ methods are explicitly implemented, to display an indicative message. According to Example 6, if the blind user presses the “Press” button, the following actions will take place: firstly, the lexical **Activated** method implementation, defined locally within the **PushButton** scheme of the **Hawk** instantiation relationship, will be executed; as a result, a virtual **Pressed** method notification will be generated, resulting in a call to the virtual method implementation of Example 6, causing the message *Pressed!* to be displayed; then, the lexical method registered in Example 6 will be called, resulting in the message *Blind user did it!* to be also displayed.

4.7 STYLE IMPLEMENTATION AND MANIPULATION

In the unified interface engineering paradigm, the various styles which result from the unified interface design process, should be mapped to implemented dialogue patterns. Developers will manipulate such implementation components in order to accomplish the desirable automatically adapted interactive behaviour. In this development process, various implementation requirements are identified, judging the appropriateness of the available tool support in facilitating style implementation and manipulation. We will formulate this set of requirements, and we will show how those requirements are met by the I-GET UIMS.

4.7.1 Implementation Requirements

Regarding style implementation and manipulation, the functional requirements are not distinguished into minimalistic or maximalistic, as it has been the case with the previously discussed development mechanisms. We define a list of requirements which must be satisfied by interface tools, if style implementation and manipulation is to be effectively enabled:

- Explicit dialogue control component class model supporting dynamic instantiation or destruction of components (i.e. based on evaluation of run-time conditions). This capability is required to allow styles *activation* or *cancellation* decisions to be easily supported. Such a requirement is met in OOP languages, where components refer to object classes, while it is also satisfied by interface builders that support some scripting facilities, while they attach programmable identifiers to interactively constructed components.
- Support for programming component instantiation or destruction in a synchronous or asynchronous manner. In this context, *synchronous* means that developers add instantiation or destruction statements as part of a typical program control flow

(i.e. via statements or calling conventions), while *asynchronous* means that the instantiation or destruction events are associated to declarative constructs such as preconditions or notifications. Normally, instantiation or destruction of components will be “coded” by developers in those points within the implementation that certain conditions dictating those events are satisfied. For this purpose, the asynchronous approach offers the significant advantage of relieving developers from the burden of algorithmically solving the problem of continuously evaluating whether those conditions, for each component class, are fulfilled during execution.

- Provision of dialogue control constructs for asynchronously *receiving* and locally *applying* adaptation decisions which are originated from external modules (i.e. independent system processes - Decision Making Component). In programming languages, this may be carried out via lower-level communication functions, while in UIMS tools, the Application Interfacing capability should offer the necessary support.
- Support for hierarchical component instantiation or destruction relationships, reflecting the hierarchical polymorphic task model. This implies that some components may be dependent on the presence of other hierarchically higher components; this reflects the need that the interface context for particular sub-tasks should be made available (to end-users), if and only if, the interface context for *ancestor* tasks is made available; for instance, the “Save file as” dialogue-box may appear, only if, the “Editing file” interface is already available to the user. We will discuss later on the isomorphic mapping of hierarchical task models to component hierarchies, showing how this promotes more the concept of design-oriented interface engineering.
- Support for orthogonal expansion of interface components. This implies that when adding new implemented styles or even interaction monitoring components, the particular structure of the control, coordination and communication software should not be affected. For instance, in conventional programming languages

which do not support declarative constructs (like preconditions, monitors and constraints), when adding interaction monitoring code to interface components, the following two steps need to be generally taken: (a) add a new monitoring activation type in the dispatcher of externally received monitoring control requests; (b) make a call for the instantiation or installation of the monitoring software, directly accessing the interaction objects of interest.

4.7.2 Style Coordination and Manipulation in the I-GET UIMS

The dialogue control component model of the I-GET UIMS is called *dialogue agents*. Agents in the I-GET language are not related to the concept of intelligent or autonomous agents, but are seen as independent threads of execution performing their own main loop, maintaining their own local state, communicating with other agents (when needed), managing arbitrary collections of interaction objects (e.g. “windows”, “menus”, “buttons”), and coordinating / controlling, or being coordinated / controlled by other dialogue agents.

4.7.2.1 Agent Classes - the Component Class Model of the I-GET Language

The implementation of dialogue agents is facilitated in the I-GET language via agent classes. Agent classes can be seen as analogous to object classes in typical OOP languages, in the sense that various instances of an agent class can be created in a dialogue program, and they constitute the component class model of the I-GET language.

In Figure 4.35, the agent class structure is outlined. There are two categories of agent classes: (a) Precondition-based agent classes, for which instances are made automatically during run-time, when their respective **on create** precondition is *satisfied* (we will explain soon the meaning of satisfaction for preconditions);

similarly, created instances will be automatically destroyed if, on the presence of destruction precondition in their respective class, the destruction precondition is satisfied. (b) Parameterised agent classes, which may be instantiated directly via instantiation statements; instantiating agent classes via statements is similar to instance creation in typical OOP languages, where agent parameters correspond to class constructor parameters. We will continue with concrete examples showing how the implementation requirements for style implementation and manipulation are met by the I-GET language facilities for defining and manipulating agent classes.

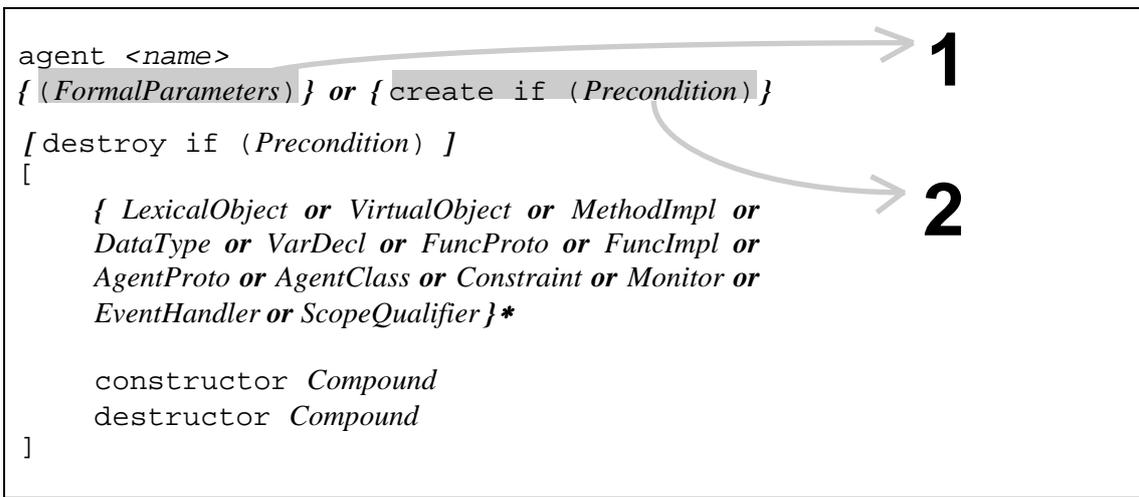


Figure 4.35 - The agent class structure: (1) the agent class header with its formal parameters, if the class is to be instantiated synchronously (i.e. via a statement); and (2) agent class header with its instantiation precondition, as well as optional destruction precondition, if the class is to be instantiated asynchronously (i.e. via preconditions).

4.7.2.2 Synchronous / Asynchronous Agent Instantiation / Destruction

In Figure 4.36, a “confirm box” sub-dialogue is implemented as an agent class component in two different ways: as a precondition-based agent class, supporting asynchronous instantiation, and as a parameterised agent class, supporting instantiation via an explicit instantiation statement. In case of precondition-based agent classes, preconditions (either for instantiation or destruction), being logical expressions, are considered to be satisfied only when there is at least one engaged

variable; the latter is assigned a value causing the precondition expression gain the **true** value. Hence, by assigning to **confirm** the value **true**, an instance of a **ConfirmBox** agent class will be automatically created.



Figure 4.36 - An agent class for implementing a “confirm box” component class, named **ConfirmBox, as: (1) precondition-based agent class; and (2) parameterised agent class. Depending on the type of an agent class, instantiation may be caused by making the precondition reach a satisfaction state (see (3), where “confirm” gets the value “true”, in any point in code), or by explicitly calling an agent instantiation (see (4), a “create” statement).**

What is more interesting is that such an assignment may be made from any point within the dialogue implementation code; hence, the developer should not care of where the **ConfirmBox** class resides, or even if such a class exists, but merely pass the **true** value to the necessary variable, i.e. **confirm**, for subsequent handling of the confirmation dialogue according to the dialogue control logic implemented within the **ConfirmBox** agent class.

In case of parameterised agents, the advantage is that re-usability is better promoted, in contrast to precondition-based agent classes, where it is evident that the instantiation precondition will have to necessarily engage some global variables; in the latter case, a specific programming context is practically assumed (i.e. a global variable of **bool** type named **confirm**), putting some regulations, though minor, for class re-usability. Also, it is clear that more code is required to manage parameterised agents; for instance, notifications originally caught from within the agent instance (i.e. button presses, via the **Pressed** method) should be propagated to the calling environment via boolean variables. The latter technique is illustrated in block 4, of Figure 4.36, where monitors are employed to grasp the changes on those boolean variables, and perform the corresponding actions. The **ConfirmBox** agent instance is capable of changing the value of those variables through their addresses, which are passed as pointer variables in its parameter set; within the methods of the confirm box buttons, the values of those variables are indirectly, i.e. via the pointer variables, changed (i.e. ***noDone=true**, ***okDone=true** and ***cancelDone=true**).

4.7.2.3 Agent Hierarchies Supporting Hierarchical Organisation of Dialogue Components

Until now, the only constructs that have been defined within the body of agent classes in the previous examples are: method implementations (i.e. callback functions), interaction objects, and the standard constructor / destructor blocks. The I-GET language allows the specification of agent classes within other agent classes (i.e.

embedded agent classes). From the syntactic point of view, this may look similar to nested classes in C++; however, in the I-GET language it has semantic implications. The specification of an agent class *X* within another agent class *Y* defines a *controlling* relationship, being different to *part-of* relationships usually applied in object hierarchies. At run-time, any newly created instance of the agent class *X* is associated to a particular instance of the agent class *Y*; the *Y* agent instance depends on the execution context which caused the creation of the *X* agent instance, as it will be explained later on. In such a runtime association, the *Y* instance is called the *parent agent instance*, while the *X* instance is called the *child agent instance*. Multiple agent instances may share the same parent agent instance, while if a parent agent instance is destroyed, then all its child agent instances are automatically destroyed by the I-GET run-time system.

```
bool NewNeeded=true;
agent DocumentProcessor create if (NewNeeded==true) [

    lexical(W95) FrameWindow win;
    lexical(W95) RadioButton font: parent={win};
    lexical(W95) RadioButton spell: parent={win};
    lexical(W95) EditControl edit: parent={win};

    agent FontStyleSet
    create if ({font}.state==RadioOn)
    destroy if ({font}.state==RadioOff) [...]

    agent SpellCheck
    create if ({spell}.state==RadioOn)
    destroy if ({spell}.state==RadioOff) [...]
]
```

Figure 4.37 - An implementation scenario in which agent hierarchies are employed.

In Figure 4.37, an example implementation scenario is provided showing how agent hierarchies can be specified. In this scenario, an interactive document processor is to be provided to the user, the dialogue implementation of which is packaged into a single agent class called **DocumentProcessor**. The document processor provides a single window which encompasses an editing area (i.e. an object instance called edit of **EditControl** lexical object class), as well as some radio-button objects, through which the presence of some windows providing miscellaneous operations can be

easily manipulated by the user. For this purpose, we show two **RadioButton** object instances, the one named **spell**, allowing the user to control the appearance of the speller dialogue box, and the other called **font**, enabling the user to control the appearance of the dialogue box for font settings. Each of those miscellaneous operations is associated with an embedded agent class, which is instantiated or destroyed on the basis of the state attribute values of its respective **RadioButton** object instance. This enables a very “clean” scheme for dialogue management, a result of the marriage of precondition-based instantiation and hierarchical agent organisation.

As is shown in Figure 4.37, the instantiation precondition of the **DocumentProcessor** agent class, **NewNeeded==true**, is initially satisfied, since the **NewNeeded** variable of **bool** type gains an initial true value. This will cause an agent instance creation, since in the I-GET run-time system, *when preconditions are installed, they are also evaluated for first time*. Hence, initially, the document processor interface will be provided to the user. If the developers wish to allow the provision of multiple instances of the document processor concurrently, then, each time a new instance of the document processor needs to be created, it suffices to assign the value **true** to the **NewNeeded** variable. The latter action will cause re-evaluation of the instantiation precondition, since a variable engaged in that precondition has been assigned with a new value (it does not matter whether the old and the new variable values are the same); from re-evaluation, the precondition will be found to be satisfied, hence, an agent instance will be created, resulting in the provision of one more document processor instance to the user.

An agent class definition embedded within another agent class is called a *child* class, while the owner agent class is called the *parent* class. For a particular agent class, all its hierarchically higher agent classes are called its *ancestor* classes. Parent-less agent classes are called *toplevel* classes. Within agent hierarchies, the engaged agent classes may be either precondition-based or parameterised; this allows powerful dialogue control schemes to be implemented.

4.7.2.3.1 Creation of Agent Instances

As mentioned, control relationships are automatically established during run-time, upon the creation of new agent instances. Each new agent instance will be assigned to a specific parent instance, depending on the execution context which caused its creation. In the I-GET language, there are two possible execution contexts which may cause agent instantiation: (a) the instantiation preconditions as such, which are handled automatically by the I-GET run-time system; and (b) the agent instantiation statements, which may be called by the dialogue developer in various points within the code. According to each execution context, which may cause the creation of a new agent instance, its parent instance is defined as follows:

- If the execution context is a precondition, then the parent instance is the one which has *installed* that particular precondition at run-time.
- If the execution context is a statement, the parent instance is the one for which the specific code block, which includes the instantiation statement recently executed, has been called.

4.7.2.3.2 Precondition Management

The simplified picture of agent instances during run-time is illustrated under Figure 4.38. Each agent instance encompasses the following items: (i) a number of members which correspond to the various items which are defined in the “body” of its respective agent class (e.g. objects, methods, variables, functions); (ii) a link to its destruction precondition (optional); (iii) links to the instantiation precondition of child agent classes (if any); (iv) a link to the parent agent instance; and (v) a list of links to all child agent instances. As it is shown in Figure 4.38, two global tables of preconditions are maintained: one for *expression-based* preconditions, and another for *API-based* preconditions. The reason for having two distinct tables, one for each precondition category, is due to the different needs for evaluating / managing those preconditions at run-time. We explain below the two different types of preconditions:

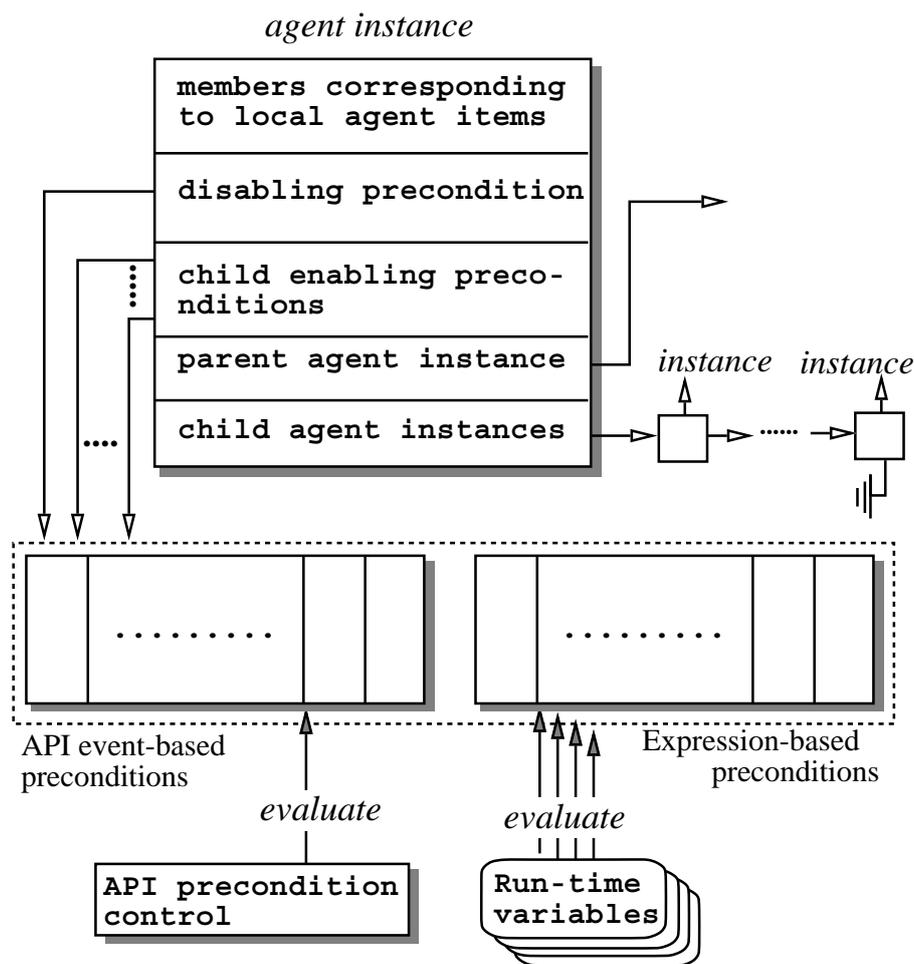


Figure 4.38 - Run-time picture of agent instances, and their connection with the precondition management strategy of the I-GET UIMS.

- *Expression-based preconditions.* These are typical programming expressions, which normally engage data variables. In the I-GET run-time system, all variables encompass a reserved call-back list for notification in case of context modification (i.e. monitors). Also, all variables maintain a “precondition engagement list”, with links to the expression-based preconditions at the global table, in which they are involved. Each time a variable changes (e.g. directly or indirectly, due to the execution of a programming statement), it notifies the precondition table to re-evaluate the necessary affected preconditions.

- *API-based preconditions.* At run-time, events from the Application Interfacing space, structured as a shared space of typed objects and as a list of typed channels, are centrally received by the I-GET run-time system, by the *API precondition control* module of Figure 4.38. Each API-based precondition in the global table holds information regarding the expected event type, being either messages or notifications for creation, modification or destruction of shared objects, as it will be discussed later on in more detail. Upon receipt of such events, the API precondition control locates the appropriate preconditions from the table which need to be evaluated.

The way in which preconditions are installed / cancelled in / from the global precondition table is defined by specific regulations defined under Figure 4.39 (those rules denote some of the standard actions that all agents perform upon instantiation and destruction). It should be noted that those details may be closer to the run-time mechanisms, rather than to the I-GET language features. However, it is believed that this information is important for dialogue developers, since it helps understand better how agents are created and destroyed upon execution; such a level of detail is also provided in mainstream programming languages like C++, since it allows developers to predict, or to justify, the effects of their implementation blocks.

1	Upon start-up all instantiation preconditions of toplevel agent classes are installed in the global table.
2	When an instance of an agent class is created, the instantiation preconditions for all of its child agent classes are installed in the global table. Its destruction precondition, is also installed.
3	When an agent instance is destroyed, the instantiation preconditions for all of its agent classes installed upon instantiation, are now cancelled from the global table. Its destruction precondition is also cancelled.
4	When an agent instance is to be destroyed, all its child agent instances are destroyed first.

Figure 4.39 - The rules for installation or cancellation of preconditions, upon agent instantiation or destruction, respectively.

4.7.2.4 Receiving and Applying Adaptation Decisions

In the unified software architecture, adaptation decisions are received asynchronously by the Dialogue Patters Component, i.e. the Dialogue Control module of the I-GET run-time system architecture, from the Decision Making Component. In the I-GET run-time architecture, the Application Component provides the link with the Decision Making Component; as a result, adaptation decisions may be received locally in the Dialogue Control module via the Application Interfacing space. The I-GET Application Interfacing space, i.e. how the Application Component and the Dialogue Control may communicate, supports both the shared space model, where arbitrary typed objects can be exposed, as well as the model of typed communication channels, which supports the establishment of an arbitrary number of strongly typed links for information exchange. The Application Interfacing (API) specification in the I-GET language involves the definition of shared data structures as well as the declaration of typed channels; representative examples are provided in Figure 4.40.

<pre> enum Command=[login, ftp, rsh, mail]; channel Commands of Command; struct Employee [string Name, Address, Role; real SalaryInDollars;] shared Employee; </pre>	1
<pre> Employee emp=["John", "Smith", "Manager",10000]; shid empId; // shared id type shcreate(Employee emp, &empId); // create shared object Commands<-login; // send a message hread(Employee &emp, shid); // read shared object </pre>	2
<pre> agent DisplayEmployee create if (shcreate(Employee emp, empId)) destroy if (shdestroy(empId)) [void Display(Employee); agent Redisplay create if (shupdate(Employee empNew, ?empId)) [constructor [Display(empNew); destroy {myagent};]] constructor [Display(emp);]] </pre>	3

Figure 4.40 - (1) Definition of a shared type and a communication channel; (2) creating a shared object, and sending a message; and (3) a precondition-based agent, with an API-based precondition.

In Figure 4.40, block 1, a single channel, named **Commands**, carrying data of **Command** type, and a single shared data type, named **Employee**, are defined. Any programmer defined data type may be engaged in defining a channel or in qualifying the type as shared. The I-GET language supplies specific statements which allow to: (i) create, modify, read and destroy shared data objects; and (ii) send typed messages through particular channels; in Figure 4.40, block 2, examples of such statements are provided. Firstly, a shared **Employee** object is created via the **shcreate** statement; this statement requires the shared object type, i.e. **Employee**, as well as a variable to hold the content, i.e. **emp**, to be explicitly supplied. Also, it requires a **shid*** variable in which a unique shared object identifier (internal type) will be automatically placed, i.e. **&empId**.

Additionally, the I-GET language supports preconditions which correspond to notifications for API-events, i.e. API preconditions as they have been previously introduced, for the following event categories: (a) creation, destruction, and modification of shared data objects; and (b) incoming messages through particular typed channels. In Figure 4.40, block 3, examples of such preconditions are provided. An agent class named **DisplayEmployee** is defined; its precondition is a **shcreate** event, which means that an agent instance will be automatically created each time an **Employee** object is placed in the shared space. The shared data type name, i.e. **Employee**, as well as the name of a variable to store the newly created shared **Employee** object content, i.e. **emp**, and the name of a **shid** variable to store the extracted shared object identifier, i.e. **empId**, have to be supplied as part of the precondition. Both implicitly declared variables, i.e. **emp** and **empId**, are made syntactically accessible within the **DisplayEmployee** agent as conventional local variables. The **DisplayEmployee** agent instance will call in its constructor the **Display** function, being a particular programmer-supplied function to display the **Employee** record; this scheme provides in a declarative manner a way to display application structures through minimal programming efforts.

In order to cope with possible modifications on **Employee** shared objects, another embedded agent is defined, named **Redisplay**, which captures modifications on displayed **Employee** shared object instances through a **shupdate** precondition; the modified content is to be locally stored in an implicitly declared variable named **newEmp**, while the **?empId** expression forces the I-GET run-time system to match the shared id of the modified object with that implicitly declared within its parent agent instance. The **Redisplay** agent calls the **Display** function with the new **newEmp** object content in its constructor, thus refreshing the displayed object with its more recent content. Then, it destroys itself; hence, the embedded agent is only needed for detecting changes on the particular shared object, and to refresh, on the presentation surface, those changes directly. This is a generic implementation strategy which can be followed, showing also the power of embedded agent classes in hiding various implementation details.

<pre>enum Decision=[Activation, Cancellation]; struct AdaptationDecision [Decision decision; string task; string style;]; channel Decisions of AdaptationDecision;</pre>	1
<pre>local AdaptationDecision *localDecisions=nil, local int totalDecisions=0; AdaptationDecision currDecision; void AddLocallyDecision(AdaptationDecision decision);</pre>	2
<pre>agent ReceiveDecision create if (message(Decisions decision)) [constructor [AddLocallyDecision(decision); destroy {myagent};] destructor []]</pre>	3

Figure 4.41 - Definitions of: (1) a channel to receive adaptation decisions; (2) local structures and functions to store adaptation decisions; and (3) an agent class for receiving and locally storing adaptation decisions.

In Figure 4.41, we provide the API definitions, as well as the implementation structure, for locally receiving adaptation decisions. This implementation module need not change, independently of the number of adaptation decisions received, but

most importantly, irrespective of the number of implemented dialogue components (as agent classes), locally within the Dialogue Control implementation. As we will show in the next Section, the locally stored adaptation decisions will be accessed within the preconditions of dialogue components, thus, enabling decisions to be automatically applied, i.e. components to get activated or cancelled, directly by the precondition management mechanisms, without imposing the need of extra code. In Figure 4.41, block 1, the definitions providing the API space for communicating decisions are supplied. In Figure 4.41, block 2, the local, to the dialogue implementation, data structures and functions, for storing adaptation decisions are provided. Finally, in Figure 4.41, block 3, the agent class to receive and locally store adaptation decisions, by utilising functions from block 2, is defined.

4.7.2.5 Orthogonal Expansion of Dialogue Components

Component expansion concerns all cases in which new dialogue components have to be implemented. We will demonstrate how the implementation of new dialogue components, being part of the Dialogue Patterns Component in a unified software interface architecture, may be carried out in the I-GET language by minimally affecting the implementation code of old dialogue components. This property is characterised as orthogonal expansion, since the software implementation of the various distinct components is made virtually independent (i.e. orthogonal) to each other; this property is also known as the principle of incremental development in the software engineering arena. In the unified development paradigm, new components are introduced for either of the following design goals:

- Extra interactive capabilities are introduced (i.e. new tasks or sub-tasks);
- Alternative styles, for polymorphic tasks, are implemented (i.e. addressing additional decision parameter values);
- Additional interaction monitoring components are implemented (for dynamic detection of user attribute values).

<pre> agent DocumentProcessor ... [... lexical(W95) RadioButton file: parent={win}; agent File create if ({file}.state==RadioOn) destroy if ({file}.state=RadioOff) [...] ...] </pre>	1
<pre> agent DocumentProcessor_AdaptivePromptingStyle create if (currDecision.decision==Activation && currDecision.task=="DocumentProcessing" && currDecision.style=="AdaptivePrompting) [...] </pre>	2
<pre> enum MonitoringStatus=[MonitoringOn, MonitoringOff]; enum MonitoringLevel=[TaskMonitoring, EventMonitoring, MethodMonitoring]; struct EventMonitoring [string objectName, eventCategory;]; struct TaskMonitoring [string task;]; struct MethodMonitoring [string objectName, methodCategory;]; struct MonitoringMsg [MonitoringStatus status; MonitoringLevel level; EventMonitoring eventInfo; TaskMonitoring taskInfo; MethodMonitoring methodInfo;]; MonitoringMsg currMessage; channel MonitoringControl of MonitoringMsg; agent MonitoringControlReceiver create if (message(MonitoringControl msg)) [constructor [currMessage=msg; destroy {myagent};] destructor []] </pre>	3
<pre> agent DocumentProcessor ... [... agent Monitoring_Object_font_Method create if (currMessage.status==<i>MonitoringOn</i> && currMessage.level==MethodMonitoring && currMessage.methodInfo.objectName=="font" && currMessage.methodInfo.methodCategory=="StateChanged") destroy if (currMessage.status==<i>MonitoringOff</i> && currMessage.level==MethodMonitoring && currMessage.methodInfo.objectName=="font" && currMessage.methodInfo.methodCategory=="StateChanged") [...] ...] </pre>	4

Figure 4.42 - (1) Adding new interactive facilities (child class); (2) adding an adaptive prompting style (toplevel class); (3) the API specifications for receiving monitoring control messages; and (4) adding a monitoring component (child class).

We will discuss some representative implementation examples, provided in Figure 4.42, which clearly show how each of the above component expansion scenarios is addressed, without, however, affecting the rest of the implementation structure. In Figure 4.42, block 1, the addition of an embedded agent class, named **File**, within the **DocumentProcessor** agent class is employed, for implementing the “File” dialogue box, commonly used in document processors (providing operations like “Open”, “Save”, “Save as...”, etc). This extra facility is also made controllable by the user via a **RadioButton** object named **file**. As it is shown, extra code is added within the original agent class, without affecting the rest of agent body implementation.

In Figure 4.42, block 2, the introduction of an additional implemented style for the “**DocumentProcessing**” task is shown, serving adapting prompting purposes, as a toplevel agent class, i.e. **DocumentProcessor_AdaptivePrompting**, while its instantiation precondition engages the **currDecision** global variable which has been defined within Figure 4.41, block 2, for storing locally the most recently received adaptation decision.

In Figure 4.42, block 3, the API specifications for receiving monitoring control messages are provided. In the unified software architecture, those messages are originated from the User Information Server; in the I-GET tool, since the Application Component offers the gateway to such external components, the monitoring messages should be communicated via the API space to the Dialogue Control module (as it was also the case with adaptation decisions coming from the Decision Making Component). The structure of messages reflects the communication protocol between the User Information Server and the Dialogue Patterns Component, as it has been defined in Section 3.3.2.2. An embedded agent class is defined in order to receive and locally store monitoring messages, named **MonitoringControlReceiver**; the

most recent monitoring control message is stored in a global variable named **currMessage**.

Finally, in Figure 4.42, block 4, the addition of a monitoring component is shown. Here, we assume that the User Information Server requests monitoring to be turned on, at the method notification level for the object “font”. The monitoring agent class, is embedded within the **DocumentProcessor** agent class, since it will have to access the **font RadioButton** interaction object. Also, it is named in a way reflecting its role, i.e. **Monitoring_Object_font_Method**, meaning it is responsible for monitoring of object **font**, at the method level. In the instantiation and destruction preconditions of the monitoring agent class access, the **currDecision** global variable is engaged. This variable holds the recent monitoring control message locally received. In the instantiation precondition, it is tested if its content is a “monitoring activation” message, while, similarly, in the destruction precondition it is tested if it is a “monitoring cancellation” message.

4.7.3 Mapping the Polymorphic Component Model to Implementation Patterns of the I-GET Language

Polymorphic task hierarchies are the central outcome of the unified interface design process. By minimising the gap between the resulting polymorphic task design and the necessary target implementation, the overall development overhead is seriously reduced, promoting potentially the quality of the resulting interactive software application. For instance, procedural decomposition could be more easily programmed through a procedure-oriented language (e.g. C or Pascal), while an Object Oriented design could be easier transformed into an implementation, if an OOP language is adopted (e.g. C++ or Smalltalk). This issue of reducing the gap among the design and implementation worlds has been recently recognised, in the interface development tool domain [Graham, 1992], as the necessity of *mapping the behavioural domain* (i.e. task design) *to the constructional domain* (i.e.

implementation artefacts), by the provision of comprehensive, well defined, engineering patterns and strategies.

From the polymorphic task decomposition process, a polymorphic hierarchy of dialogue components is produced. In such a component hierarchy, for any pair of components, we distinguish the following two cases: (a) they both belong to the same set of alternative styles for supporting a particular polymorphic task: and (b) the components concern different, either polymorphic or unimorphic, sub-tasks. For each of these two cases, we will identify the set of possible design relationships among the components, which will have to be preserved within the target implementation, so that the polymorphic component hierarchy, i.e. the unified interface design, can be thoroughly mapped into a target implementation form. In this context, we will firstly identify those design relationships, and we will subsequently provide the engineering strategy through which those relationships can be easily mapped to implementation patterns in the I-GET language. As a result, we would have established a mapping of the polymorphic component hierarchy into appropriate implementation patterns, thus establishing effectively the final link in the chain bridging the unified interface design “world” and the implementation “world” of the I-GET development tool.

Components belong in the same set of alternative styles for a polymorphic task	Components correspond to different, polymorphic or unimorphic, tasks
<ul style="list-style-type: none"> • Exclusion 	<ul style="list-style-type: none"> • Containment
<ul style="list-style-type: none"> • Substitution 	<ul style="list-style-type: none"> • Aggregation
<ul style="list-style-type: none"> • Augmentation 	<ul style="list-style-type: none"> • Collaboration
<ul style="list-style-type: none"> • Compatibility 	<ul style="list-style-type: none"> • Control
	<ul style="list-style-type: none"> • Indifference

Figure 4.43 - Relationships between components which belong in the same set of alternative styles for a polymorphic task (left Table), and relationships among components which correspond to different tasks (right Table).

In Figure 4.43, the two categories of relationships among components defined in a polymorphic task hierarchy are outlined. The relationships for components being

alternative styles of the same sub-task (left Table of Figure 4.43), have been defined and explained in the discussion of the unified interface design process. The definition for the second group of relationships, regarding pairs of components which are associated to different tasks, follows.

- *Containment.* One component is physically contained into the other; for instance, the editing windows of Microsoft Word™ are physically enclosed within a single top-level window of the Microsoft Word™ application.
- *Aggregation.* Both components are direct physical constituents of another component, with which they are both related via a containment relationship. For instance, the “Drawing toolbar” and the “Formatting toolbar” of Microsoft Word™ are direct constituents of the Microsoft Word™ main application window.
- *Collaboration.* When one component exposes information or services to the other in a “read only” fashion (i.e. “clients” cannot alter such information, while services offered do not affect the behaviour of the “server”). For instance, monitoring components access the interaction objects residing in the physical components for which they have to collect interaction monitoring information.
- *Control.* One of the components exposes information or services to the other in a “write” fashion. This means that various behavioural or presentational aspects of the “server” component may be changed by a “client” component by calling those services or by changing the accessed information. For example, in Microsoft Word™, the user is able to turn -on or -off the appearance of certain toolbars, via a pop-up menu displayed when the right mouse button is clicked over any toolbar; logically, the component responsible for this pop-up dialogue should have “write” access to the “visibility” attribute of those toolbar components.
- *Indifference.* The components are simply not related to each other. For instance, we do not expect any relationship among the “Print” dialogue box and the “Drawing toolbar” of Microsoft Word™.

Containment	The container supplies an appropriate parent object instance to the contained component, to establish containment appropriately at the object hierarchy level.
Aggregation	The constituent components share the same parent object instance, supplied by their common container component.
Collaboration	Public agent members, i.e. variables and functions, which are made available by collaborating components, not affecting the owner behaviour or presentation, are used by their “partner” components.
Control	Public agent members, i.e. variables and functions, which are made available by “controlled” components, affecting the owner behaviour or presentation, are used by the “controller” components.

Figure 4.44 - Mapping relationships among components which correspond to different tasks into implementation patterns of the I-GET language.

The mapping of those relationships to I-GET language constructs is very simple and straightforward; it should be noted that such a mapping is also effectively supported in typical OOP languages. However, on the one hand we have to show that indeed this necessary implementation facility is offered by the I-GET tool, while on the other hand, we certify that it is supplied together with all the rest of the mechanisms for style implementation and manipulation, like declarative activation and orthogonal expansion, which are not directly supported in 3rd generation programming languages. In Figure 4.44, the corresponding implementation patterns in the I-GET language, for each of those relationships, are identified.

4.8 ARCHITECTURAL OPENNESS

The need for architectural openness emerges due to the following key engineering properties of unified interfaces: (i) they are distributed software systems, composed of multiple processes communicating with each other; (ii) they encompass various

dialogue components, which may utilise interaction elements from multiple toolkits. In this context, we have identified four fundamental architectural schemes, each reflecting specific architectural needs of unified interfaces:

- *The gateway scheme.* In this scheme, the original interactive system architecture is vertically expanded towards the unified software interface architecture, without affecting the original architectural links. The idea is to locate an appropriate component, of the original architecture, which may play the role of a gateway to the components of the unified architecture. In the I-GET tool, there are two alternative possibilities: (a) communication via the API space, i.e. the API gateway; and (b) connection via the Dialogue Control component, employing some powerful hooking capabilities of the I-GET language, i.e. the Dialogue Control gateway.
- *The distributed dialogue components scheme.* In this scheme, a blending of the component-based and the distributed computing paradigms is employed for dialogue patterns. The idea is to have dialogue patterns as distributed software objects, which may inter-operate, at run-time, in the context of a single unified distributed interactive application. We will show how the I-GET tool, through its Generic Toolkit Interfacing Protocol (GTIP) complies to this scheme.
- *The toolkit interoperability scheme.* This has been already discussed, and concerns the need for promoting the creation of cross-toolkit object hierarchies, leading to interactive components in which elements from various toolkits are mixed. We will show how the I-GET language, as well as the toolkit server architecture, are open to support toolkit interoperability.
- *The hardware-free software scheme.* This scheme reflects the need of enabling unified interface implementations to run across a wide range of target equipment and machinery, without being distracted by external factors such as absence of peripheral devices or operational device failures. We will show how the toolkit integration architecture of the I-GET tool allows dialogues to run safely, even

when some software or hardware resources are missing at the end-user terminal; this is a property not met in even dynamic multi-platform languages, like Java™.

4.8.1 The API Gateway for Architectural Extensions

In the API gateway technique, the contact point between the unified interface architecture and the I-GET run-time architecture is the Application Component; hence, the Application Component pays the role of a *proxy* component. Consequently, the communication among the User Information Server and the Decision Making Component, with the Dialogue Control module (playing the role of the Dialogue Patterns Component), is necessarily facilitated through the Application Interfacing space.

In order to implement this type of connection effectively, it is important to enable the implementation of some standard re-usable code, through which potentially any interactive application built with the I-GET tool can be directly expanded towards a unified software architecture; this should be facilitated by simply encompassing the extra software modules in the original implementation, without requiring particular modifications. In this context, we have identified two points in which such re-usable extra software is needed:

- At the Application Component side, where the communication with the external modules should be implemented. Any appropriate communication software, ranging from sockets to higher-level communication packages, may be employed. In the communication with the components of the unified architecture, the semantics of the inter-component communication protocol, as defined in Section 3.3, should be obeyed.
- At the Dialogue Control module, where the indirect communication, via the proxy component (i.e. Application Component), with User Information Server and the

Decision Making Component should be implemented, relying upon the API communication facilities.

In Figure 4.45, the API gateway architecture extension scheme is illustrated. The extra code to be written at the Dialogue Control side, is code in the I-GET language, similar to the communication-specific examples shown for receiving and locally storing adaptation decisions (see Section 4.7.2.4), as well as for receiving and storing monitoring control messages (see Section 4.7.2.5).

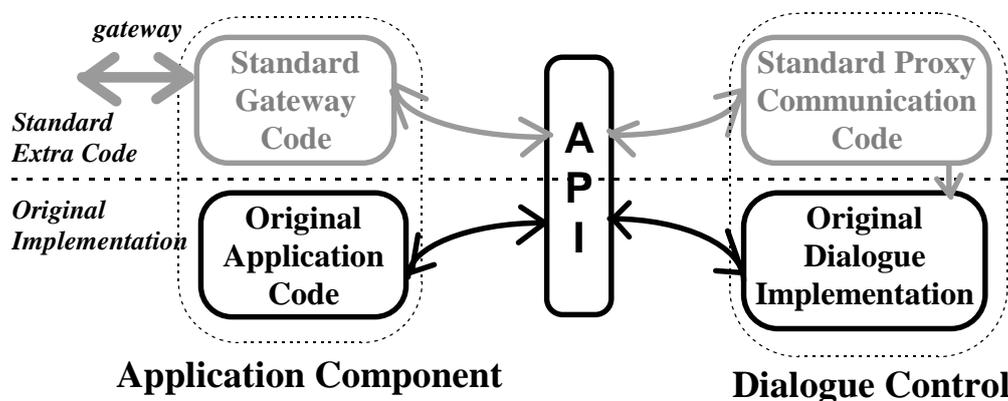


Figure 4.45 - Orthogonal expansion of an I-GET application, towards a unified interface architecture, through the API gateway.

4.8.2 The Dialogue Control Gateway

In the Dialogue Control gateway scheme, the idea is to expand appropriately only the Dialogue Control module, primarily for communication purposes. Such an extension implies that the dialogue implementation facilities of the I-GET language, offer all the required development constructs to implement the communication with external processes, other than those of the I-GET run-time architecture. Hence, in simple terms, the *standard gateway code* package of the API gateway technique is now moved directly into the Dialogue Control implementation. This is supported in the I-GET language via a powerful model for mixing with C++, a capability going beyond conventional language hooks, as they are supported by most programming languages.

Following the I-GET language mixing model, developers may inject C++ communication management code within the I-GET dialogue implementation code, at various points, ranging from global declarations to statements, while exchange of data between I-GET and C++ code is effectively facilitated through *bridge* statements, which allow: (a) I-GET expressions to be assigned to C++ variables; and (b) C++ expressions to be assigned to I-GET variables.

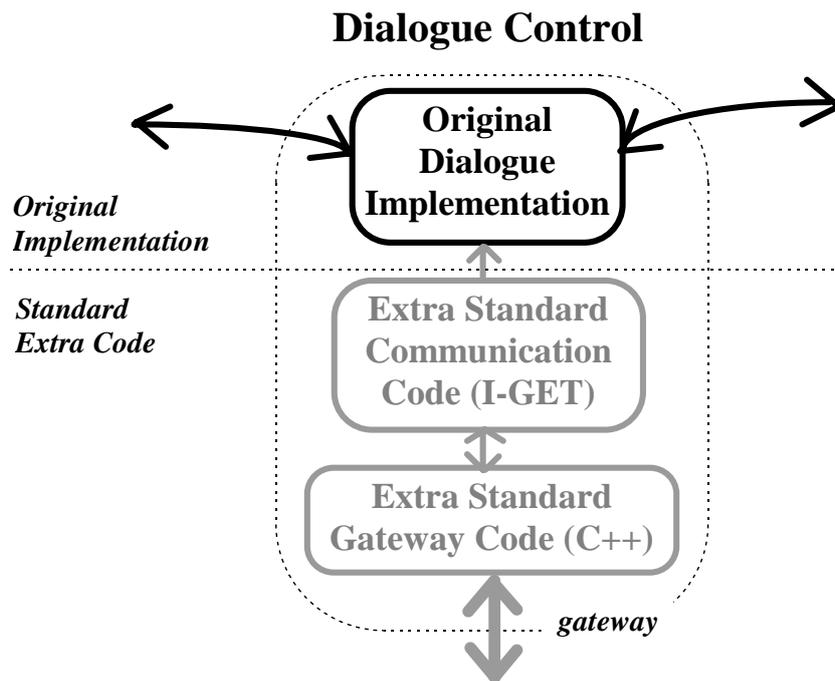


Figure 4.46 - Orthogonal expansion of an I-GET application, towards a unified interface architecture, through the Dialogue Control gateway.

In Figure 4.46, the Dialogue Control gateway scheme is outlined. Again, some I-GET code has to be written, for storing locally the received messages into I-GET dialogue variables, either for adaptation decisions or for monitoring control messages. The translation from C++ data structures, as they will be communicated by the *extra standard gateway code (C++)*, to local I-GET language data variables, is to be implemented as part of the *extra standard communication code (I-GET)*, via bridge statements.

4.8.3 Distributed Component-Based Interfaces

The notion of component-based development has been previously discussed, mainly attributing to the principle of binary-level re-use of implemented parts of software systems. Technologies like CORBA and OpenDOC™ are well known as successful component-ware software infrastructures. In the context of interface development, components may refer to implemented design patterns, associated with particular domain-specific sub-tasks. Styles, as they have been defined in the framework of the unified interface design method, constitute perfect candidates for interface components.

In the I-GET language, there is an explicit powerful component model, though not supporting binary-level re-use. This (binary-level re-use) is, not only beyond the objectives of this thesis, due to the large additional development overhead, but it also presents some evident technical difficulties: (i) there are more than one candidate technologies, like OpenDOC™, JavaBeans™, DCOM™ and ActiveX™, and CORBA; (ii) those technologies hardly inter-operate; and (iii) most of them are still subject to change and they have not been yet stabilised.

Apart from componentisation, there is another emerging issue for interactive applications, which is tightly coupled with the idea of componetware, being that of distributed execution. Distribution and comonentisation are so much related, that in many cases there is misunderstanding as to when a particular technology is targeted in supporting distributing computing or component-based development. For instance, CORBA is only a distributed technology, while JavaBeans is only a component technology.

The key difference is that, while distributed computing technologies offer a model for breaking down software into concrete components which may offer services accessed in a *remote* fashion, component-technologies are mainly based on breaking down the interactive aspects of applications into meaningful binary code pieces, which may be

combined altogether into a single software package. One key commonality is that both technologies rely upon “programming interfaces” of software components, thus enabling alternative software components to be employed, as far as they conform to the particular programming interfaces expected.

Up until now, we have seen only one research effort to marry distributed computing and component-ware together, potentially promoting distributed component-based interfaces: the Fresco User Interface System [X Consortium, 1994]. In Fresco, InterViews interaction objects have been provided as CORBA objects, supporting development of interactive applications which may run in a fully distributed fashion, as a number of various running inter-operating processes, each handling a specific part of the interface. There are two possible architectural patterns for such approaches, aiming to support distributed object computing techniques in interface development. Those patterns reflect two possible alternative levels of distribution:

- *Interaction objects.* In this case, interaction object instances created may physically execute their own dialogue control in different host machines. As a result, interface developers may implement dialogues in which the object instances engaged may run on different machines, while the physical interface should always run on the user-terminal host. This approach requires communication protocols among distributed interaction objects, for: (i) *space negotiation*, since container objects should allocate screen space for contained objects via inter-process communication; and (ii) *event distribution*, since events should be propagated appropriately from container objects to the contained objects, via inter-process communication. Distribution at the level of interaction objects necessitates radical changes on the toolkit implementation structure.
- *Dialogue components.* In this case, the various implemented dialogue patterns may execute in a distributed fashion, while interaction objects may be served by one specific object server; normally, the object server will run on the user-terminal host. In order to support distributed dialogue component implementation, the following key functional requirements are identified: (i) provision of the *toolkit*

library as a single remote service, engaging various distributed object classes, supporting object naming services (i.e. locating object identifiers via unique names); (ii) provision of *dialogue components as distributed objects*. The distributed components approach requires no changes on the original toolkit library, hence, being more easy for application, while it reflects more realistic distribution requirements, when compared with distribution of interaction objects.

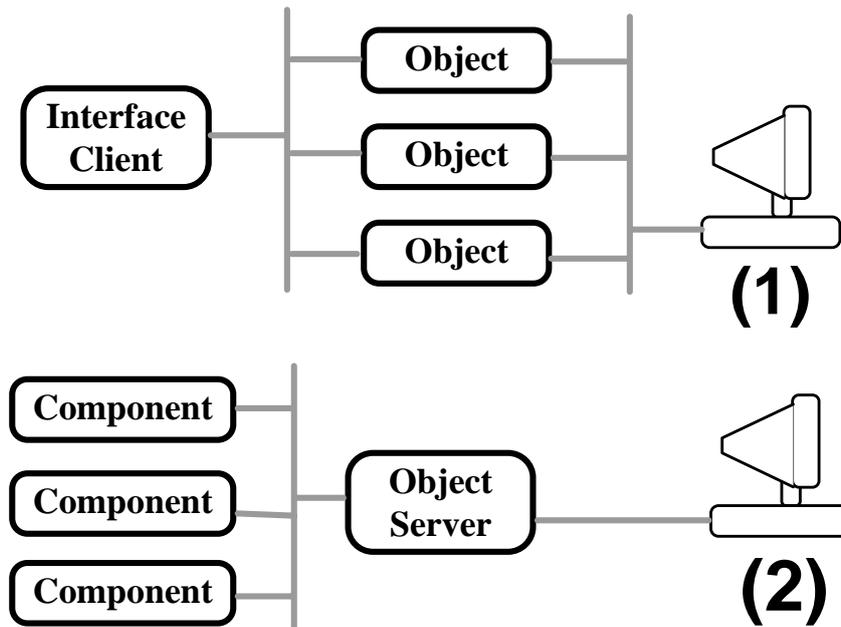


Figure 4.47 - (1) The *distributed toolkit* architectural scheme; and (2) the *distributed interface* architectural scheme.

In Figure 4.47, the architectural schemes for the two distribution approaches previously discussed are illustrated. The distribution at the level of interaction objects is mentioned as the *distributed toolkit* scheme, since it practically results in a distributed library of interaction objects. The component distribution approach is mentioned as the *distributed interface* scheme, since the various components which execute as remote objects are those which constitute the dialogue management logic. The distributed interface scheme is not only the easiest to implement, but also it reflects more realistic software distribution demands, either in terms of execution efficiency (distribution for small software entities, turns the gain from parallel execution, to loss, due to communication overhead), as well as in terms of reusability

requirements (interaction objects, being the “smallest pieces”, turn to be standardised, being available in all development platforms, while components, being the pre-fabricated “mechanical parts”, represent a resource from which dialogue developers will choose on a need-to-use basis).

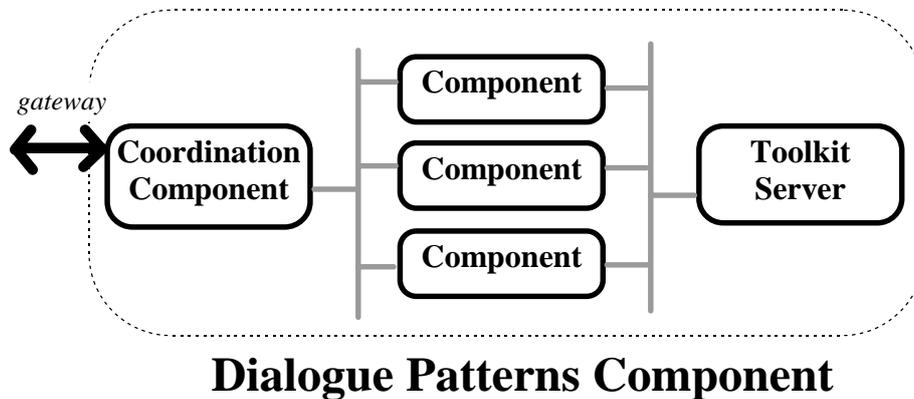


Figure 4.48 - Blending the distributed component paradigm with the unified interface development architecture.

In the I-GET UIMS, the toolkit server role matches the object server architectural component, of the distributed interface scheme (see Figure 4.47, part 2). The role of the toolkit server in the I-GET UIMS is to separate the physical binding of interaction elements, from the dialogue control logic, by offering the Generic Toolkit Interfacing Protocol (GTIP), being actually a meta-programming interface, for communication among object servers and remote clients. The I-GET UIMS does not provide built-in support for distribution at the level of dialogue control components, though some extensions could be built as standard dialogue libraries (i.e. not affecting the I-GET UIMS system software as such); such possible future extensions will be briefly discussed in Chapter V.

When trying to blend the component distribution paradigm with the unified interface development approach, the need for a central coordination component is identified, playing a two-fold role: (a) it offers a gateway with external components, i.e. User Information Server, Context Parameters Server and Decision Making Component; and (b) it locates and controls the various dialogue components, according to the

adaptation decisions received from the Decision Making Component. As a result, the combination of this coordination component, with the distributed components and the central object server, realises the Dialogue Patterns Component of the unified software architecture (see Figure 4.48).

4.8.4 Toolkit Interoperability

Toolkit interoperability is defined as the capability to construct cross-toolkit hierarchies. The implementation infrastructure for supporting toolkit interoperability varies from customised changes on specific toolkits, in order to allow their interoperation, towards open generic protocols among toolkits for object naming, space negotiation, and event propagation. One issue in toolkit interoperability is that of the programming model; since more than one toolkits may be employed, it is evident that there are more than one candidate programming models, corresponding to each of the interoperating toolkit libraries. In this context, by programming model we mean the implementation style for: creating / destroying object instances, accessing object attributes, constructing object hierarchies, adding / removing call-backs, adding / removing event handlers, etc. Normally, it is hard to meet different toolkits providing the same programming model. Some approaches like that of Xt, for X Windowing System, aimed at achieving the goal of a single programming model, however, real practice has proved that independent toolkit vendors did not follow this paradigm (e.g. OSF/MOTIF, Xaw/Athena, InterViews, Xview).

In the I-GET language, the single programming model is supported, and the interaction elements from all imported toolkits are manipulated through the same implementation facilities of the I-GET language. Additionally, in the I-GET language, the default behaviour is to facilitate the construction of cross-toolkit hierarchies, i.e. allowing the parent object to belong in a toolkit different from that of the child object, while such cross-toolkit hierarchies have to be supported by their respective toolkit server implementations. If the developer wishes to activate compile errors for cross toolkit hierarchies, the `-parentlayersame` compile flag has to be used. In Figure

4.49, the idea of merged toolkit servers is shown, when cross-toolkit hierarchies are to be supported. A merged toolkit server is practically a combined server implementation for the various inter-operating toolkits, since in writing toolkit server code for serving GTIP requests originated from the Dialogue Control module, interaction elements from the various inter-operating toolkits will have to be utilised.

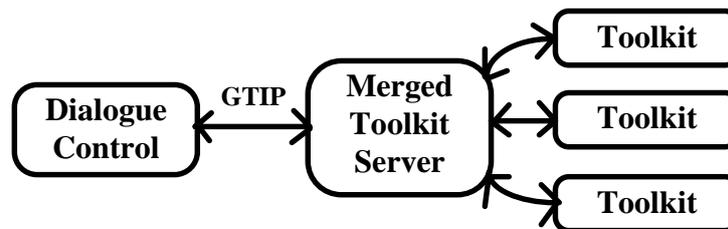


Figure 4.49 - The merged toolkit server paradigm for importing inter-operating toolkits in the I-GET UIMS.

4.8.5 Multi-Platform Interactive Software

Implementing software which may run on various hardware / software platforms is a largely demanding task. Various parameters have to be taken into consideration, one of which is the existence of various categories of alternative peripheral equipment. In this context, the notion of “multi-platform” concerns those software layers of unified interactive software which handle the various I / O peripheral devices, as well as those which establish the link to the software interaction elements available at the target machine. Multi-platform graphical toolkits are too restrictive in this sense, since they are merely targeted in supporting windowing interaction in graphical displays, with the presence of a mouse and / or a keyboard. Development efforts towards software which may run on diverse hardware and software platforms have led to the Java™ language. However, there are two key technical issues in such software development approaches:

- *Does “run anywhere” mean “run in the same manner” as well ?* It is well known that when developing a Java™ application, even though it may run on different platforms, it still does not display in the same way. This is not only attributed to the presence of alternative interaction elements on different platforms (e.g. Windows, Motif, or Macintosh objects), but also to differences on the “imaging model”, i.e. the graphical display model for drawing graphics and text, which causes inherent deviations on the appearance of interface objects. As a result, carefully calculated spatial relationships may not be displayed in the same manner on different platforms, while, inherently, the intended aesthetic result may be seriously affected.
- *Is the software able to “recognise equipment”, or shall we build “alternative software versions” ?* This question concerns all cases where the various different platforms provide radically different I / O support. For instance, one platform might provide a character-based terminal with only a few lines of display space, while another platform might support a 3D-auditory area. In Java™, alternative software versions will have to be built, since, for each target platform, a different version of the language core libraries is supplied. This approach is a simple way out of a potential operational failure; however, it introduces some software engineering problems, due to the maintenance of replicated (i.e. common) code across the distinct application versions.

In the I-GET language, there are specific ways to tackle the above two central issues, which are, arguably, the most appropriate for the unified interface development demands. More specifically, regarding the requirement for running in the same manner, or at least in a way reflecting all the various aesthetic design principles, the I-GET language supports the separate implementation of the physical design aspects, for each different target toolkit. Following this practice, developers may implement unified applications in which the physical dialogue components are certified to perfectly match the properties of the target toolkit graphical display policies. It is likely that, this will “sound” to be a step back, considering that multi-platform toolkits had actually emerged in order to eliminate the replication overhead due to distinct

physical interface implementations. The key clarifying point here is that, in the I-GET language, common implementation is still supported via virtual interaction objects, while the added-value capability introduced, is that specialisation of physical interactive features can be implemented by directly accessing the distinct toolkit-specific object instances, through the unified virtual object instances.

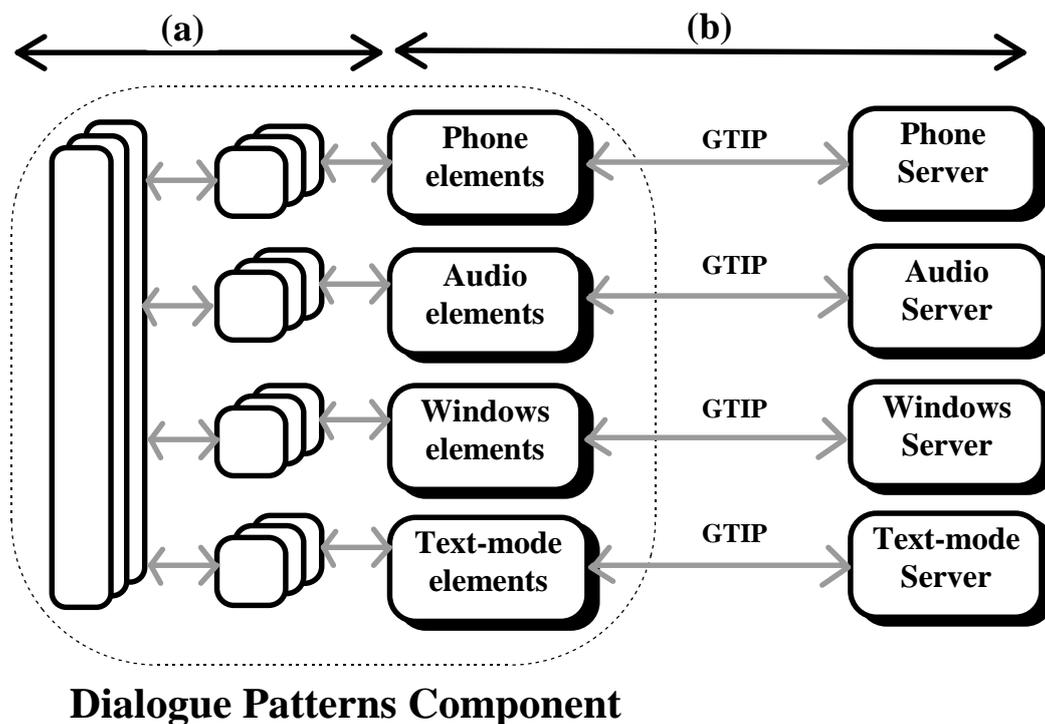


Figure 4.50 - The multi-platform interface development approach in the I-GET UIMS: (a) due to support for specialisation of physical properties; and (b) due to toolkit servers.

On diverse hardware / software platforms, the presence of different input / output peripherals and software interaction elements, may cause potential operational problems in interactive software products. Such operational problems are eliminated when interactive applications are built using the I-GET tool: Firstly, the physical binding of interaction elements and input / output peripherals in the I-GET UIMS is realised at the toolkit server side. In other words, the Dialogue Control module never accesses directly any platform-supplied software or hardware resource. Secondly, when running a unified application to a particular platform, only the toolkit server(-s) for that platform need to execute; this will have the effect that only those dialogue

components which encompass interaction elements served by the particular running toolkit server(-s) will be physically realised, while the rest will not cause any operational error.

In Figure 4.50, we show the two key properties for multi-platform unified software: (a) the capability to specialise dialogue implementation for various platforms / toolkits, still supporting common abstract implementation (see the Dialogue Control module); and (b) the ability to isolate the programming of physical resource management at the toolkit server side (i.e. the various alternative toolkit servers of Figure 4.50), while making the respective physical interaction elements fully available for dialogue development, through the toolkit integration capability (i.e. the “elements” boxes in the Dialogue Control module of Figure 4.50).

Chapter V

SUMMARY, CONCLUSIONS AND FUTURE WORK

5.1 DISCUSSION

In Chapter IV, the desirable functional properties of interface tools have been identified, in order to support the employment of the unified interface development paradigm. Those requirements have emerged primarily due to the need for unifying and managing together implemented dialogue patterns, which reflect diverse user attribute values, as well as usage-context characteristics. Hence, we can briefly describe the role of unified implementation practices as being directed towards *managing diversity*. Additionally, if we study separately the development needs of the various distinct dialogue patterns as such, isolated from the unification perspective, i.e. *implementing diversity*, it is naturally expected that typical dialogue construction functional requirements will apply. It is evident that both, managing diversity, as well as implementing diversity, will have to be substantiated in the context of a unified interface development process.

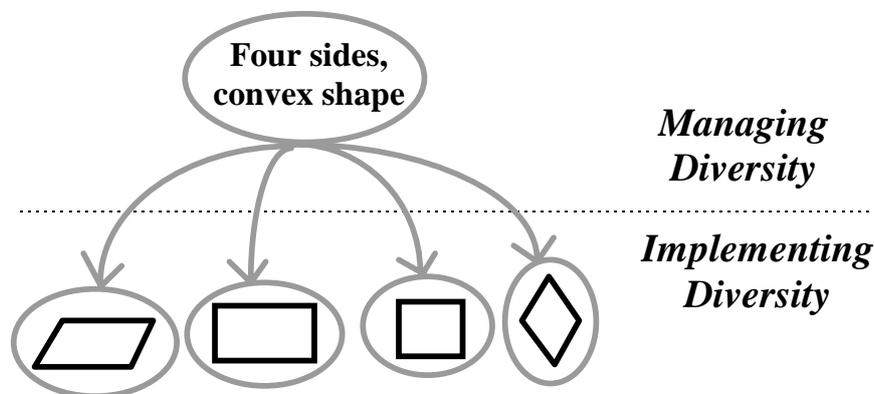


Figure 5.1 - Managing diversity versus implementing diversity in a simple example taken from geometry.

The relationship between these two fundamental categories of development processes is illustrated in Figure 5.1 with an example from geometrical shapes. Managing diversity implies the capability to unify the physical properties of visually diverse structures into a single unified mathematical formula; even though all the unified artefacts are well represented by the abstract relationships, specific differentiated

techniques have to be applied for drawing each distinct geometric shape. We will briefly study existing prevalent interface development practices by assessing their appropriateness for managing diversity. The main weak or strong points of such development methods, when recruited for unified development, will be identified. Finally, we will conclude by organising development methods in two general groups: one which concerns those methods which are considered as more appropriate for managing diversity, and the other which encompasses those methods being judged as less appropriate for this role.

Presentation-based	Constructing the appearance of the artefact.
Physical-task based	Decomposing physical user-actions.
Demonstration-based	Demonstrating physical user-actions.
State-based	Implementing dialogue-state transition logic.
Event-based	Implementing input-event reaction logic.
Declarative 4GL methods	Implementing around “when”, as opposed to “how”.
Model-based	Defining various models (e.g. dialogue, data).
Abstract objects / components	Manipulating artefacts relieved of physical issues.
Abstract-task based	Decomposing “what” the user will do, but not “how”.
Semantic-based	Defining only the semantic information and services.

Figure 5.2 - Categories of interface development practices.

There are numerous dialogue building techniques, as well as interface tool categories. A comprehensive survey can be found in [Myers, 1995]. For the purposes of tool assessment, we have compiled a list of the most prevalent practices for building and manipulating dialogue artefacts, according to the type of implementation facilities offered to interface developers; this list is provided in Figure 5.2.

As it has been already discussed, in the unified interface development paradigm, the key ingredients in managing diversity are abstraction and encapsulation. As a result, development methods closer to the physical level of interaction are not appropriate for manipulating and coordinating diverse dialogue artefacts. For instance, through graphical construction techniques (i.e. *presentation-based*), a specific graphical interface instance is constructed; as a result, it is required to employ alternative dialogue control facilities, other than graphical construction methods, as well as to

construct alternative interface components, in order to address diverse design parameters. Hence, graphical construction by itself suffices for implementing diversity; however, it is not adequate for managing diversity. Examples of tools supporting presentation-based techniques are: MENULAY [Buxton et al, 1983], being one of the first interface builders; LUIS [Manheimer et al, 1990]; TAE Plus™ [TAE Plus, 1998]; and VisualBasic™.

Physical-task based techniques lead to one particular decomposition of user-actions, i.e. unimorphic decomposition, thus not leaving enough space for the design and implementation of alternative task structures, i.e. polymorphic decomposition, as it is potentially required when designing for diverse user attribute values. Examples of such techniques are: TAG [Reisner, 1981], being one of the first task-based dialogue specification methods; and UAN [Hartson et al, 1990].

Demonstration-based techniques allow the interactive definition of a desirable physical interface, in an example-based fashion. In this sense, those techniques still fall in the category of graphical interface construction methods, since the end-result is one particular physical interface instance. Examples of tools supporting demonstration-based techniques are: PERIDOT [Myers, 1988], being the first system to support example-based methods; Pavlov [Worlber, 1996]; TRIP3 [Miyashita et al, 1992]; and DEMOII [Fisher et al, 1992].

State-based techniques are not necessarily tight to any particular development style, since they constitute generic fundamental computable models. However, state models do not incorporate any abstraction facilities as such, but merely provide an implementation framework for algorithmically representing the dialogue control logic. Such implementation models are less popular today due to the large development overhead, inherent from the fundamentally primitive nature of the basic implementation constructs (i.e. states, transitions, and basic I/O). Examples of state-based techniques are: StateCharts [Wellner, 1990], and STN [Jacob, 1988].

Event-based techniques are for interactive systems what state-based techniques are for general computation systems (i.e. programs). Even though it is possible to implement abstraction via events, such as logical input, the technique seems to be more appropriate for intra-component dialogue control requirements, such as handling physical user-input within a specific dialogue context, e.g. drag-and-drop dialogues, as opposed to inter-component dialogue control requirements, that is needed for coordinating alternative implemented interface components. The Sassafras UIMS [Hill, 1986] has been an event-based UIMS, while the event model is supported by all known interface toolkits.

Development techniques which are farther to the physical level of interaction, focusing on higher-level dialogue properties, are generally more appropriate for addressing dialogue diversity needs. For instance, *declarative 4GL methods*, supporting precondition- / notification- based schemes for activating sub-dialogues, are considered as very good implementation models for coordinating alternative dialogue patterns. Examples of tools supporting declarative 4GL methods are: ViewControllers in the SERPENT UIMS [Bass et al, 1990]; dual dialogue agents in the HOMER UIMS [Savidis et al, 1995a]; and agent classes in the I-GET UIMS [Savidis et al, 1997f].

Development approaches which rely upon *abstract interaction objects or dialogue components*, by supporting alternative physical realisations, are very promising for unified development, because of their built-in implementation support for polymorphism at the physical interaction level. Examples of methods supporting abstractions of interaction objects are: meta-widgets [Blattner et al, 1992]; and virtual objects [Savidis et al, 1995a].

Similarly, techniques supporting *abstract tasks*, enable the construction of user dialogues not being bound to a single hard-coded task model, as it is the case with physical task decomposition. Abstract task models will normally have to be instantiated as concrete physical task structures. When alternative instantiations are allowed to be embedded within the implementation, facilitating run-time coordination

and selection, we make a big step towards the unified interface development paradigm. We do not know of any tools which support the notion of user-task abstraction.

Semantic-based techniques are very promising, even though the present support demonstrates a very narrow view of the capabilities which could be integrated within a semantic-based development process. Today, from semantic specifications, mainly attributing to the type of functional services / information intended to be interactively supplied to the user, a single-minded physical design is directly produced by semantic-based development tools. There is a need for generating intermediate dialogue layers, supporting design parameters which the developer may control in order to affect the type of dialogue artefacts produced; in this way, semantic-based methods may serve as design generators, still requiring methods for coordination and manipulation of those artefacts.

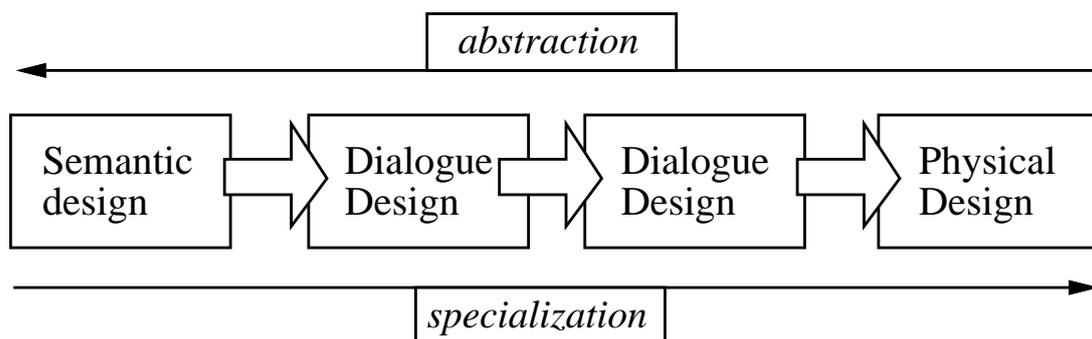


Figure 5.3 - Generating intermediate design layers in semantic-based methods.

In Figure 5.3, the distinctive role of semantic-based methods in automated artefact production is indicated, while various levels of abstraction (or specialisation, when seen in the opposite direction) are needed, according to various design parameters controlled by developers. Following this perspective, even though semantic-based methods do not contribute directly in managing diversity, they are considered as a very promising method for unified development, due to their fundamental design-

oriented interface engineering policy. Examples of tools supporting semantic-based development are: Mickey [Olsen, 1989]; and UofA [Singh, 1989].

Finally, *model-based techniques* are, theoretically, the most promising category of interface tools, since they potentially incorporate computable user- and design-models. However, in most existing model-based tools, the emphasis has been shifted from user-modelling towards application modelling and dialogue modelling. As a result, the interface generation policies have been centred around the problem of “closing the gap between the application model and the dialogue model”, rather than “closing the gap between the user model and the dialogue model”. Some of the model-based tools do include some heuristics design rules, on the basis of which the dialogue model is incrementally constructed according to hard-coded design rules, mapping specific categories of application structures to particular classes of dialogue objects. This type of functional behaviour still falls in the context of design-oriented interface engineering, however, it is far away from adaptable interface generation, engaging primarily user- and usage-context design parameters.

Today, another disadvantage of model-based tools is their monolithic nature. It is believed that the emphasis should be shifted towards model-based development in an interdisciplinary fashion, allowing the employment of diverse tools, each serving a specific distinctive purpose in interface development, while tools should be connected together on the basis of a common architectural vision, concrete development model, and protocols for information exchange. Examples of model-based tools are: UIDE [Foley et al, 1991]; and Mobi-D [Mobi-D].

Following the study of the various interface development methods, we reach the general conclusion that those techniques which are targeted in producing artefacts being relatively closer to the physical level of interaction, are generally less appropriate for managing diversity. However, development methods which support manipulation of artefacts which do not directly engage lexical interaction issues, are in

an advantageous position for addressing the needs of unified development. These remarks lead to the separation of the various development practices in two categories, as indicated in Figure 5.4.

Promising for managing diversity	Not appropriate for managing diversity
<ul style="list-style-type: none">• Declarative 4GL methods.• Abstract objects / components.• Model-based.• Abstract-task based.• Semantic-based.	<ul style="list-style-type: none">• Presentation-based.• Physical-task based.• Demonstration-based.• State-based.• Event-based.

Figure 5.4 - Classifying development practices in two groups: those being promising for managing diversity (left column), and those considered as inappropriate for this role (right column).

5.2 FUTURE WORK

In this thesis, a new interface engineering paradigm has been presented, i.e. the unified interface development paradigm, aiming to address the challenge of creating interfaces for “all”, being automatically adapted to individual user needs, as well as to the particular context of use. Also, the technical profile of interface tools for supporting unified development has been drawn, while a tool particularly developed for supporting unified interface implementation, the I-GET UIMS, has been discussed. In a unified interface development process, as in any dialogue development effort, there are various stages, specific roles for each stage, as well as numerous resource categories which need to be manipulated by persons involved in such roles. The unified paradigm poses some extra requirements in the various dimensions of an interface development process, necessitating further research and development efforts. We will provide a list of milestones to be further achieved, in order to effectively promote the unified engineering paradigm with better development support; this list will be provided by means of a research agenda, being the next step, having as a starting point the results produced in the context of this thesis.

5.2.1 New Interaction Technologies, Metaphors and Toolkits

The need for development efforts leading to new interaction methods, interaction metaphors and interface toolkits has been already identified in the context of tool requirements for unified development, under Section 4.2. The importance of such efforts is particularly emphasised, since, the construction of dialogue artefacts for diverse user attribute values and contexts of use, should rely upon the appropriate available interaction technologies. For instance, auditory dialogues may be structured, only if the required audio-based interaction techniques are present.

During the early testing stages of the I-GET UIMS, one significant difficulty has been the absence of diverse interaction technologies, in order to assert the capability of the I-GET UIMS for managing diversity. This, among other reasons, has led us in designing and implementing alternative interaction libraries, such as the HAWK non-visual toolkit [Savidis et al, 1997c], and the SCANLIB library of witch-based scanning techniques [Savidis et al, 1997b].

5.2.2 Design Tools for the Unified Design Method

The outcome of the unified interface design process is a hierarchical structure of tasks and interface components, in which alternative decompositions, i.e. polymorphism, are supported. Additionally, decision parameter values and design documentation is also attached to the designed artefacts. The development of tools to support collaborative design activities, providing the necessary range of facilities for manipulating design artefacts, such as: creating, storing, querying, retrieving, documenting, discussing, etc, may considerably help designers to effectively and efficiently manage the demanding unified interface design process.

5.2.3 Reusable Components of the Unified Software Architecture

In the unified software architecture, the various components may be implemented as expandable generic services, shared by many applications in different domains. For instance, the User Information Server could be developed as a central repository of user profiles, while it may also encompass inference mechanisms for detecting dynamic user attributes, on the basis of interaction monitoring. Running applications could connect to such a server for user authentication, profile extraction and for user attribute detection.

Similarly, the Decision Making Component may be also supplied as a central server, encompassing design information and adaptation-oriented decision making for multiple application domains.

For both the User Information Server and the Decision Making Component, the problem of execution efficiency may emerge when multiple running applications are connected to them. This problem will emerge in adaptivity-oriented behaviour, requiring continuous run-time communication, rather than on adaptability-oriented behaviour, where decisions are made only at start-up, before initiation of interaction. For this reason, the software to handle adaptivity-oriented adaptations may “close” and “migrate” in the local environment of each running application. Such facilities, like “cloning” and “migration”, are supported in existing tools, like in the Voyager™ system [Voyager, 1997].

5.2.4 Extension of the I-GET UIMS for Distributed Dialogue Components

The I-GET UIMS generates code which is packaged as a single process. Currently, what “disables” I-GET from supporting the construction of different parts of an interactive application as distinct running Dialogue Control components, is that one

executing instance of each necessary (to the particular dialogue implementation) toolkit server, should be initiated for each running Dialogue Control module. As it has been previously discussed in section 4.83, inter-operation among running components requires a central toolkit / object server, shared by the independent running components comprising a single distributed interactive application, so that resource sharing can be enabled (e.g. exchange of object identifiers, common object naming service). It should be noted that, the technical decision to make the I-GET run-time system initiate one server for each single application has been dictated for the following two reasons:

- In some toolkits, the toolkit server applications, being also conventional client-programs for those toolkits, may only create one top-level application window (e.g. in Xt); as a result, the toolkit server cannot serve more than one dialogue applications, since, the latter would require each one top-level window to be created, while the toolkit server would be allowed to make only one such instance;
- The toolkit server has to support fast responding schemes for requests originated by the Dialogue Control module (via GTIP); in case that multiple applications have to be server, the execution efficiency is severely damaged.

The extension of the I-GET UIMS for distributed execution may rely upon the CORBA distributed object computing technology. There are three main extensions required: (a) CORBA-based toolkit servers should be built, for all target toolkits imported (as it has been done in Fresco [X Consortium, 1994]); (b) the toolkit interface specification of all respective toolkits should be expanded with appropriate C++ hooks and bridges to generate calls to the CORBA toolkit server; and (c) the initiation sequence of the I-GET run-time library should be provided with an additional version for CORBA (a very small part of the I-GET run-time library). This approach is illustrated in Figure 5.5.

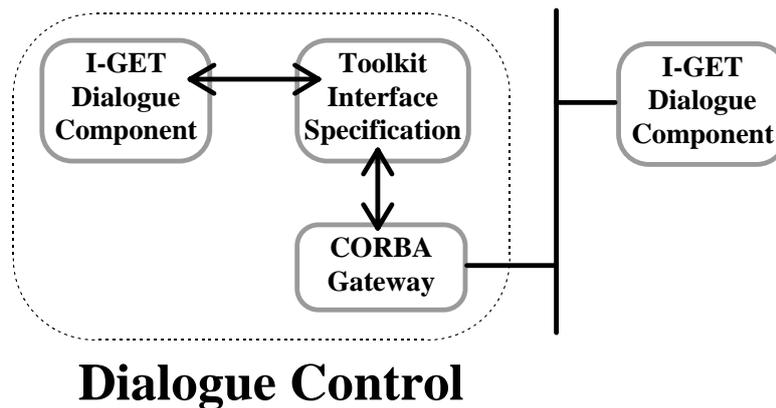


Figure 5.5 - Extending I-GET to support distributed dialogue components.

5.3 SUMMARY AND CONCLUSIONS

The need for unified interface development emerges when trying to construct an appropriate technical framework in order to address the “User Interfaces for All” objective. As it is shown in Figure 5.6, the starting point of this trajectory, which logically leads to the unified interface development paradigm, is that diverse user attributes and different contexts of use will have to be taken into consideration in the design of interactive software applications and services “for all” (i.e. step 1). Those varying user- and usage-context- attribute values establish different design requirements, which naturally affect the design of dialogue artefacts (i.e. step 2). As a result, alternative dialogue artefacts will have to be constructed, at various points of the interface design process, as dictated by the differing user- and usage-context- attribute values (step 3). When trying to map the outcome of such diversity-originated design processes into an implemented interactive application, a key issue is how the various alternative dialogue artefacts will be packaged.

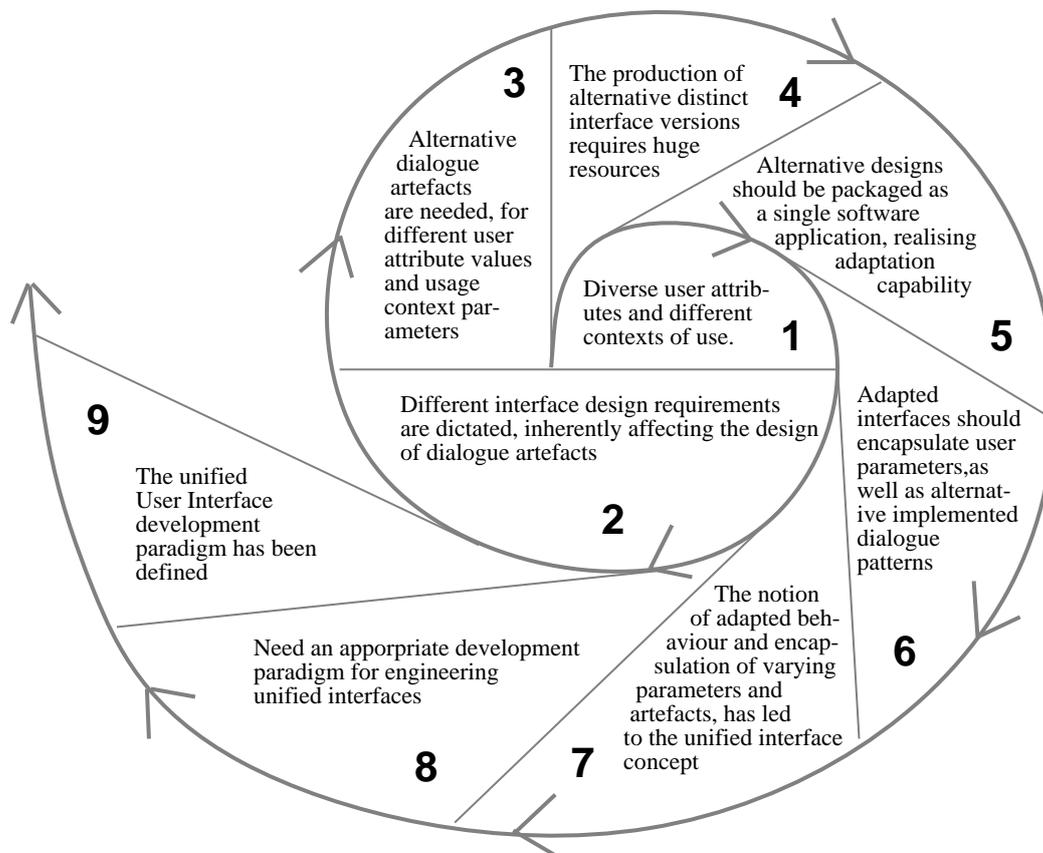


Figure 5.6 - From diverse users and usage contexts, towards the unified interface engineering paradigm.

The production of alternative interface versions requires huge resources for production, maintenance, upgrade and distribution, and turns to be practically unrealistic, since, all potential distinct versions should be made available for execution (i.e. step 4). This is particularly evident in the case of non-desk-top computer systems, where the number of end-users is unrestricted.

Consequently, the packaging of the various alternative dialogue patterns into a single software application is considered as the most promising approach; in this context, packaging may not necessarily imply the construction of a monolithic software system encompassing all the various dialogue artefacts, but could be also instantiated

as logical collection into a single resource, i.e. a repository, made directly accessible by a single software application which encompasses an adaptation capability, i.e. being able to choose the most appropriate dialogue patterns for a particular end-user and target usage-context (i.e. step 5).

In order to facilitate such an adaptation capability, adapted applications should encompass information about individual users, as well as alternative dialogue patterns in an implemented form (i.e. step 6); this notion of encapsulation, has led to the definition of the Unified User Interface concept, realising adapted user- and usage-context- behaviour, by encapsulating all the various varying design parameters and alternative dialogue artefacts (i.e. step 7). The need for an appropriate development strategy for unified interfaces (i.e. step 8), has led to the design of the unified interface engineering paradigm (i.e. step 9), which has been introduced and presented in detail, in the context of this thesis.

The unified interface engineering paradigm consists of the unified interface design method, and the unified interface implementation method. There is one common “vision” between these two development stages, the unified software interface architecture. Even though an architecture as such has been traditionally considered to be mostly linked with the implementation domain, more recent software engineering trends emphasise the explicit visualisation of various high-level aspects of an architectural system plan to the design space [Thomas et al, 1995]. A similar approach is applied in constructing buildings. The detailed architecture, which may contain information varying from cartographic or thermodynamic analysis, to data regarding materials and electricity installations, is primarily targeted to the construction engineers; however, the external view of the architecture is useful to future habitants for appropriately designing their home.

The unified development paradigm emphasises a design-oriented interface engineering approach. Concepts, entities and relationships from the design domain are directly preserved in the target unified implementation. For instance, styles, and user-tasks are mapped to implemented dialogue components which should be locatable, at

run-time, on the basis of the respective style and user-task names. Additionally, the notion of activation or cancellation adaptation decisions, being defined as part of the adaptation design logic, are explicitly supported in the implementation domain as adaptation messages, causing instantiation or destruction of the necessary dialogue components. This type of direct mapping of design items to implementation structures promotes the effective communication among the design and implementation worlds. Also, the outcome of the unified interface design method (a polymorphic structure of user-tasks and interface components), provides by itself a design model very close to the implementation world.

We have used the example of object hierarchies to demonstrate how the elimination of the design-implementation gap may reduce the development cycle, while in the mean time, promoting the quality of the resulting dialogues. More specially, designers “think” in terms of object classes, while they design by physically arranging objects (i.e. *containment*). On the implementation side, programmers still manipulate object classes, while they construct object hierarchies in which the *parenthood* relationship corresponds to physical containment. As a result, on the one hand programmers may directly transform an object-based design into a target implementation, while on the other hand, designers may easily understand object hierarchies; hence, the communication among designers and programmers may become largely more effective and fruitful.

The unified design method, as well as the unified implementation paradigm, have been defined to be open to existing strategies and methodologies. More specifically, the unified interface design method provides a model for organising adaptation-oriented dialogue artefacts, in which their design relationships, as well as the respective adaptation-specific decision parameters are explicitly represented. Hence, the role of the unified design method is to support designers in *managing design for diversity*, while particular design practices may be freely employed by interface designers, when the various dialogue patterns have to be physically designed and documented. In a similar manner, the unified implementation method provides an architectural framework, in which, roles, control flow, implementation requirements,

and communication protocols are explicitly defined; however, no particular programming technique or tool is imposed. Additionally, the distributed nature of the architecture allows different development methods to be employed for each distinct architectural component.

When a tool developer or a tool user is confronted with the issue of unified development, it is necessary to be aware of the most significant functional requirements, in supporting unified interface implementation. In this context, we have identified and discussed seven key functional properties and mechanisms which interface tools should satisfy, in order to support effectively and efficiently the engineering of unified software interfaces: metaphor development, toolkit integration, toolkit augmentation, toolkit expansion, toolkit abstraction, style manipulation and coordination, and architectural openness.

We have also introduced and discussed the I-GET UIMS, an interface development tool encompassing implementation mechanisms particularly suited for unified development. Additionally, we have positioned existing prevalent interface development practices according to their appropriateness for supporting a unified interface implementation process, a role we have generally characterised as *managing diversity*. From such an assessment, it is concluded that the most promising development methods for managing diversity are those being less close to the physical implementation world; however, the distinctive role of development techniques being more close to the lexical level of interaction is also acknowledged, in order to support target implementation of the various designed dialogue artefacts, characterising this latter role as *implementing diversity*.

In Figure 5.7, the models of three typical dialogue development processes are indicated, regarding the time spent for each different level of the User Interface. In model 1, emphasis is given mainly on physical design; this is a typical scenario for most interactive software products, since one particular interface design instance is implemented (i.e. the “averaging” approach, as it has been discussed in Chapter I). In model 2, a “normalised” approach is shown, i.e. one in which all levels are considered

to have equal importance. This model could be a good starting point for promoting unified interface development, since physical design, even when it is instantiated in a unimorphic fashion, may be more easily altered or expanded, since it relies upon the syntactic and semantic design layers.

Finally, model 3, represents our desirable target, in which design is mainly focused on semantic issues, while the transition towards the dialogue and physical levels may be facilitated by appropriate design generators, collecting components from interface design pattern repositories. In order to reach such a level of maturity, component technologies should be technologically encapsulated, i.e. become a de-facto software infrastructure, while dialogue components for various work-tasks should be released by component manufacturers.

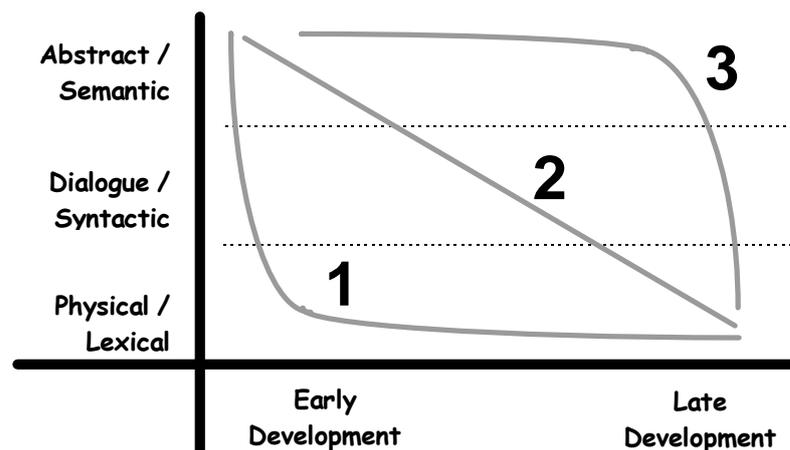


Figure 5.7 - Three general development process models: (1) emphasis on physical design; (2) equal emphasis on each level; and (3) emphasis on abstract design.

A similar situation, in which technology evolved in an analogous manner, has been observed with interaction toolkits; in the beginning of the 80s, interface toolkits have been mainly research systems, while after one decade, on the one hand they became a standard software development layer, while on the other hand, some particular sets of interaction objects dominated and became de-facto commercial standards. It is believed that the introduction of a unified development approach for adapted interactions establishes one route towards the “User Interfaces for All” objective;

however, it also opens the road for further research and development, since there are many open challenging questions which have to be further addressed (see Figure 5.8):

- In unified development, one top-level issue is how do we identify those computable varying parameters which affect the way in which people interact with machines. In other words, we have to *expose diversity* in those human properties which are likely to dictate alternative dialogue means.

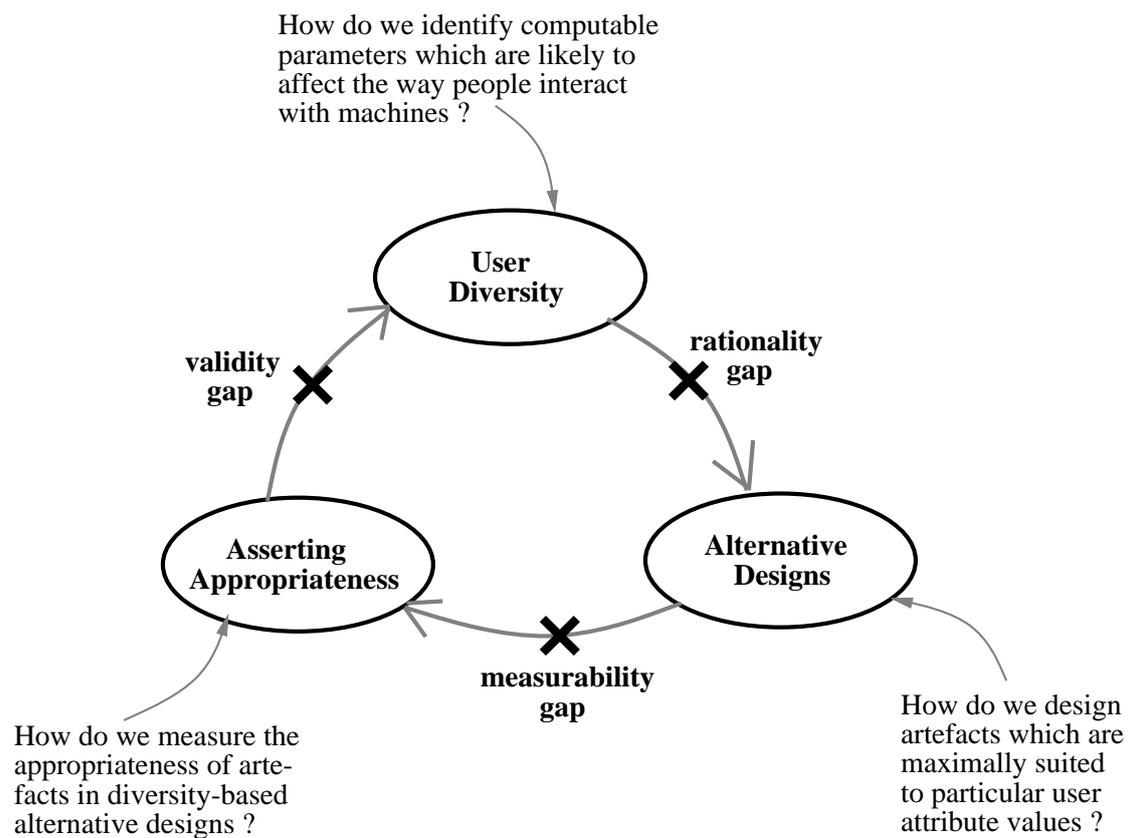


Figure 5.8 - Open questions in designing adapted behaviours.

- Even when the varying parameters are discovered, how do we design dialogue artefacts which are intended to maximally match those diverse attribute values? Hence, we have to *implement diversity*, relying upon appropriate design rationale clearly relating diverging parameter values with specific properties of the target

dialogue artefacts. Our incomplete knowledge about performing effectively the transition from user diversity towards alternative design artefacts is characterised as the *rationality gap* (see Figure 5.8).

- Finally, in solving the equation of how to structure appropriate alternative patterns for diverse user attribute values, we need to be able to assert that the resulting designs do serve their intended goals; i.e. thus support accessible and high-quality interaction. Such a process requires appropriate evaluation methods, and the capability to measure appropriateness of designed artefacts. At present, this is argued to be still a missing link, characterised as the *measurability gap* (see Figure 5.8) If we are able to assert the appropriateness of artefacts, then we practically validate the produced design solutions towards user diversity; hence, if we are unable to carry out such an evaluation process, the *validity gap* is created (see Figure 5.8).

The unified interface engineering paradigm is an inter-disciplinary process driving the production of automatically adapted software applications and services. It is intentionally not very prescriptive, so that it does not exclude particular design and implementation practices, while it is descriptive enough to drive the engineering process of unified software systems. As any new development paradigm, it naturally requires some initial investment to be effectively assimilated and applied; however, if the constructed software products are intended to be used by diverse user populations, operated in different usage contexts, it is strongly believed that the gains will overcome the expended resources.

At the beginning of an Information Society era, there is already a clear trend towards an integrated computerised way of manipulating machinery, for all sorts of human activities. In such a progress, an amazing combination of interaction techniques has been realised, with cross-domain impact. For instance, computerised TVs, TV interfaces to computers, Web phones, interactive cartoons, etc, are all representative examples of such developments. This evolution has been technically enabled through

the following strategy: *hardware becomes less specialised, more interoperable, removing various hard-coded operations and components, while software is written more-and-more, to replace such originally hardwired functions.* This strategy opens the road towards automatic adaptations, since, while hardware had to be used “as it is”, software may “on the fly” be adapted, as needed. In view of the emerging technological infrastructure, the unified interface development paradigm provides a new and promising technical framework to structure adapted software products, which are open, expandable, as well as compliant to the recently popular development paradigms, namely distributed object computing, and component-based development.

BIBLIOGRAPHY

[ACCESS D151, 1995]. Stephanidis, C, Savidis, A., Gogoulou, R. Report on the integration of target platforms. Deliverable D.1.5.1, June 1995, © ACCESS Consortium 1995.

[ACCESS D132, 1995]. Stephanidis, C., Savidis, A. Final report on G-DISPEC. Deliverable D.1.3.2, December 1995, © ACCESS Consortium 1995.

[ACCESS Project, 1996]. The ACCESS Project - Development Platform for Unified Access to Enabling Environments. RNIB Press, September 1996.

[ACCESS D142, 1997]. Stephanidis, C., Savidis, A. Final report on the I-GET tool. Deliverable D.1.4.2, January 1997, © ACCESS Consortium 1997.

[Adam et al, 1997]. Adam, N., Awebuch, B., Slonim, J., Wegner, P., Yesha, Y. Globalizing Business, Education, Culture, Through the Internet. *In CACM Journal Vol 40 (2)*, 115-121.

[Akoumianakis et al, 1996]. Akoumianakis, D., Savidis, A., Stephanidis, C. An Expert User Interface Design Assistant for Deriving Maximally Preferred Lexical Adaptability Rules. In *Proceedings of the 3rd World Congress on Expert Systems*, Seoul (Korea), 5-9 February 1996, 1298-1315.

[Akoumianakis et al, 1997]. Akoumianakis, D., Stephanidis, C. Knowledge-based Support for User-adapted Interaction Design. *Expert Systems with Applications*, Vol 12(2), pp. 225-245, 1997.

- [Asaw, 1997]. <http://www.screenaccess.com/asawinfo.html>.
- [Bangemann, 1994]. Recommendations to the European Council: Europe and the global information society - The Bangemann Report, Brussels, Belgium. Available at: <http://www.ispo.ccc.be/infosoc/backg.bangeman.html>
- [Barkakati, 1991]. Barkakati, N. Model-View-Controller (MVC) Architecture of Smalltalk-80. In *Object-Oriented Programming in C++*. SAMS Publishing, 1991, Indiana, USA, 74-85.
- [Bass et al, 1990]. Bass, L., Hardy, E., Little, R., Seacord, R. Incremental development of User Interfaces. In *Engineering for Human-Computer Interaction*. G. Cockton, Ed. North-Holland, 1990, 155-173.
- [Belloti, 1993]. Belloti, V. Integrating Theoreticians' and Practitioners' Perspectives with Design Rationale. In proceedings of the INTERCHI'93 conference on Human Factors in Computing Systems (April 24-29), Amsterdam, Netherlands, 101-106.
- [Benyon et al, 1988]. Benyon, D. MONITOR: A Self-Adaptive User-Interface. In *Human-Computer Interaction - IFIP INTERACT'84, Volume 1*, Elsevier, 1984, 307-313.
- [Berners-Lee, 1997]. Berners-Lee, T. World-Wide Computer - the Human Connection. In *CACM Journal Vol 40 (2)*, 57-58.
- [Blanchard, 1997]. Blanchard, H. User Interface Standards in the ISO Ergonomics Technical Committee. In *SIGCHI Bulletin, Volume 29 (1)*, ACM, 1997, 20-24.

- [Blattner et al, 1992]. Blattner, M., Glinert, E., Jorge, J., Ormsby, G. Metawidgets: Towards a theory of multimodal interface design. In *proceedings of the COMPSAC'92 conference*, Chicago (September 22-25, 1992), IEEE Computer Society Press, 115-120.
- [Browne et al, 1990a]. Browne, D., Totterdell, M. , Norman, M. (Eds). Adaptive User Interfaces, Academic Press, London, 1990, 195-212.
- [Browne et al, 1990b]. Browne, D., Norman, M., Adhami, E. Methods for Building Adaptive Systems. In *Adaptive User Interfaces*, Browne, D., Totterdell, M., Norman, M. (Eds). Academic Press, London, 1990, 85-130.
- [Buxton et al, 1983]. Buxton, W., Lamp, M., Sherman, D., Smith, K. Towards a comprehensive User Interface Management System. In *Computer Graphics*, July 1983.
- [Canfield et al, 1982]. Canfield, Smith, D., Irby, D, Kimball, R., Verplank, B., Harlsem, E. Designing the Star User Interface. In *Byte Magazine*, April 1982.
- [Card et al, 1983]. Card, S., Moran, T., Newell, A. The psychology of Human-Computer Interaction, Hillsdale, NJ, Lawrence Erlbaum, 1983.
- [Card et al, 1987]. Card, S., Henderson, D. A multiple, virtual-workspace interface to support user task switching. In proceedings of the ACM CHI+GI'87 Conference on Human Factors in Computing Systems and Graphics Interfaces, Carroll, J., Tanner, P. (Eds), 1987, 53-59.
- [Carroll et al, 1988]. Carroll, J., Mack, R. L., and Kellogg, W. A. *Interface Metaphors and User Interface Design*. In Handbook of Human-Computer Interaction. M. Helander (ed). Elsevier, 1988, 67-85.

- [Cockton, 1987]. Cockton, G. Some Critical Remarks on Abstractions for Adaptable Dialogue Managers. In *proceedings of the Third Conference of the British Computer Society, People & Computers III, HCI Specialist Group*, University of Exeter, September 7-11, 1987, 325-343.
- [Cockton, 1993]. Cockton, G. Spaces and Distances - Software Architecture and Abstraction and their Relation to Adaptation. In *Adaptive User Interfaces*, Schneider-Hufschmidt, M., Kuhme, T., Malinowski, U. (Eds), North-Holland, 1993, 79-108.
- [Cote, 1993]. Cote Munoz, J. AIDA - An Adaptive System for Interactive Drafting and CAD Applications. In *Adaptive User Interfaces*, Schneider-Hufschmidt, M., Kuhme, T., Malinowski, U. (Eds), North-Holland, 1993, 225-240.
- [Coutaz, 1990]. Coutaz, J. Architecture Models for Interactive Software : failures and trends. In *Engineering for Human-Computer Interaction*. G. Cockton, Ed. North-Holland, 1990, 137-151.
- [Cowan et al, 1993]. Cowan, D., Durance, C., Ciguere, E., Pianosi, G. CIRL/PIWI: A GUI Toolkit Supporting Retargetability. *Software-Practice and Experience*, Vol 23 (5), May 1993, 511-527.
- [Desoi et al, 1989]. Desoi, J., Lively, W., and Sheppard, S. Graphical Specification of User Interfaces with Behavior Abstraction. In *Proceedings of the CHI'89 Conference on Human Factors in Computing Systems* (Austin, Tex, April 30-May 4, 1989), 139-144.
- [Dieterich et al, 1993]. Dieterich, H., Malinowski, U., Kuhme, T., Schneider-Hufschmidt, M. State of the Art in Adaptive User Interfaces. In *Adaptive User Interfaces*, Schneider-Hufschmidt, M., Kuhme, T., Malinowski, U. (Eds), North-Holland, 1993, 13-48.

- [Duke et al, 1993]. Duke, D., Harrison, M. Abstract Interaction Objects. *Computer Graphics Forum*, 12 (3), 1993, 25-36.
- [Duke et al, 1994]. Duke, D., Faconti, G., Harrison, M., Paterno, F. Unifying view of interactors. *Amodeus Project Document: SM/WP18*, 1994.
- [Edwards, 1995]. Edwards, A. Computers and people with disabilities. In *Extra-Ordinary Human-Computer Interaction - Interfaces for Users with Disabilities*. Edwards, A. (Ed), Cabridge University Press, 1995, 19-43.
- [Edwards et al, 1994]. Edwards, K., Mynatt, E. An Architecture for Transforming Graphical Interfaces. In *proceedings of the 7th Annual Symposium on User Interface Software and Technology (UIST'94)*, Maria del Rey, November 2-4, 1994, 39-48.
- [Fayad et al, 1996]. Fayad, M., Cline, M. Aspects of Software Adaptability. In *CACM Journal*, 39 (10), October 1996, 58-59.
- [Fisher et al, 1992]. Fisher, G., Busse, D., Wolber, D. Adding Rule-based Reasoning to a Demonstrational Interface Builder. In *proceedings of the ACM symposium on User Interface Software and Technology (UIST)*, 1992, 89-97.
- [Foley et al, 1983]. Foley, J., Van Dam, A. Fundamentals fo interactive computer graphics. Addison-Wesley Publishing, 1983 (1st edition), 137-179.
- [Foley et al, 1984] Foley, J. D., Wallace, V. L., Chan, P.. The human factors of computer graphics interaction techniques. *IEEE Computer Gr. & Appl*, 4, 11 (November 1984), 13-48.

[Foley et al, 1991]. Foley, J., Kim, W., Kovacevic, S., Murray, K. UIDE - An Intelligent User Interface Design Environment. In *Architectures for Intelligent Interfaces: Elements and Prototypes*, Sullivan, J, Tyler, S. (Eds). Addison-Wesley, 1991.

[Gamma et al, 1995]. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[Goldberg et al, 1984]. Goldberg, A., Robson, D. Smalltalk-80: The Interactive Programming Environment. Addison-Wesley Publishing, 1984.

[Graham, 1992]. Graham, T. Future research issues in languages for developing User Interfaces. In *languages for developing User Interfaces*, Myers, B. (Ed), Jones and Bartlett Publishing, 1992, 401-418.

[GUIB Project, 1995]. Textual and Graphical User Interfaces for Blind People. The GUIB PROJECT. Public Final Report. RNIB Press, 1995.

[Guinan, 1997]. Guinan, J. Platform-Independent C++ GUI Toolkits. In *C/C++ Users Journal (CUJ)*, January 1997, 19-26.

[Gus, 1997]. <http://www.access-by-design.com/wwwboard/messages/99.html>.

[Hartson et al, 1989]. Hartson, R., Hix, D. Human-Computer Interface Development: Concepts and Systems for its Management. *ACM Computing Surveys*, 21(1), March 1989, 241-247.

[Hartson et al, 1990]. Hartson, H. Rex., Siochi, Antonio. C., and Hix, Deborah. The UAN: A User-Oriented Representation for Direct Manipulation Interface Design. *ACM Trans. Inform. Syst.* 8, 3 (July 1990), 289-320.

- [Hill, 1986]. Hill, R. Supporting Concurrency, Communication and Synchronisation in Human-Computer Interaction - The Sassafras UIMS. *ACM Trans. Gr.* 5(3), July 1986, 289-320.
- [Hoare, 1978]. Hoare, C. A. R. Communicating Sequential Processes. In *Commun.* ACM 21, 8 (Aug, 1978), 666-677.
- [INCUBE, 1997]. <http://www.commandcorp.com/cci/enhanced.html>.
- [Jacob, 1988]. Jacob, R. An executable specification technique for describing Human-Computer interaction. In *Advances in Human-Computer Interaction, 1*. Hartson, R (Ed). Ablex Publishing, New Jersey, 1988, 211-242.
- [Jacobson et al, 1997]. Jacobson, I., Griss, M., Johnson, P. Making the Reuse Business Work. In *IEEE Computer*, October 1997, 36-42.
- [JavaSoft, 1997]. Java: coming soon to phones, TV set-tops, hand-held devices. <http://java.sun.com/features/1997/oct/Personal.Embedded.html>. JavaSoft, 1997.
- [Johnson et al, 1988]. Johnson, P., Johnson, H., Waddington, P, Shouls, A. Task-related knoweldge structures: analysis, modelling, and applications. In Jones, D. M.; Winder, R. (Eds), *People and computers: from research to implementation*, Cabridge University Press, 1988, 35-62.
- [Kline et al, 1995]. Kline, R., Glinert, E. Improving GUI Accessibility for People with Low Vision. In proceedings of the ACM CHI'95 Conference on Human Factors in Computing Systems, May 7-11, Denver, Colorado, 1995.
- [Kobsa et al, 1989]. Kobsa, A., Wahlster, W. *User Models on Dialog Systems* (Eds), Springer, Berlin, 1989, 4-34.

- [Kobsa, 1990]. Kobsa, A. Modelling the user's conceptual knowledge in BGP-MS, a user modelling shell system. *Computational Intelligence* 6, 1990, 193-208.
- [Kouroupetroglou et al, 1996]. Kouroupetroglou, G., Viglas C., Anagnostopoulos, A., Stamatis, C., Pentaris, F. A Novel Software Architecture for Computer-based Interpersonal Communication Aids. In *Proceedings of ICCHP'96 - 5th International Conference on Computers Helping People with Special Needs*, July 17-19, Linz, Austria, 1996, 715-720.
- [Kurzweil, 1992]. Kurzweil, R. *The Age of Intelligent Machines*. MIT Press, 1992, 39-39.
- [Manheimer et al, 1990]. Manheimer, J, Burnett, R., Wallerns, J. A case study of user interface management system development and application. In *proceedings of the CHI'90 conference on Human Factors in Computing Systems* (Seattle, Washington, April 1-5, 1990), ACM.
- [Marcus, 1996]. Marcus, A. Icon Design and Symbol Design Issues for Graphical Interfaces. In Del Galdo, E., and Nielsen, J. (Eds), *International User Interfaces*, John Wiley and Sons, New York, 1996, 257-270.
- [Markopoulos, 1997]. Markopoulos, P., Johnson, P, Rowson, J. Formal aspects of task based design. In *proceedings of the 4th Eurographics Workshop on Design, Specification and Verification of Interactive Systems*, Granada (Spain), June 4-6. Harrison, M., Torres, J. (Eds). Springer-Verlag, Wien, 1997, 209-224.
- [McLean et al, 1995]. McLean, A., McKerlie, D. Design Space Analysis and Use-Representations. Technical Report EPC-1995-102, Rank Xerox, 1995.

- [Miyashita et al, 1992]. Miyashita, K., Matsuoka, S., Takahashi, S., Yonezawa, A., Kamasa, T. Declarative Programming of Graphical Interfaces by Visual Examples. In proceedings of the ACM symposium on User Interface Software and Technology (UIST), 1992, 107-116.
- [Mobi-D]. Stanford Univ. Knowledge Systems Laboratory, The Mobi-D Interface Development Environment,
<http://www-SMI.Stanford.EDU/projects/mecano/mobi-d.html>.
- [Myers, 1988]. Myers, B. Creating User Interfaces by Demonstration. Academic Press, Boston, 1988.
- [Myers, 1990]. Myers, B. A new model for handling input. *ACM Trans. Inform. Syst.* 8, 3 (July 1990), 289-320.
- [Myers, 1995]. Myers, B. User Interfaces Software Tools. In *ACM Trans. on Human-Computer Interaction*, 12(1), march 1995, 64-103.
- [Mynatt et al., 1994]. Mynatt, E., Weber, G. Nonvisual Presentation of Graphical User Interfaces: Contrasting Two Approaches. In Proceedings of the CHI'94 conference on Human Factors in Computing Systems (Boston, Massachusetts, April 24-28, 1994), 166-172.
- [Mynatt et al., 1995]. Mynatt, E., Edwards, W. Metaphors for non-visual computing. In *Extra-Ordinary Human-Computer Interaction - Interfaces for Users with Disabilities*. Edwards, A. (Ed), Cabridge University Press, 1995, 201-220.
- [Nielsen, 1997]. Nielsen, J. User Interface Design for the Web. All day tutorial #1, HCI International'97, August 24, HCI International'97, San Francisco.
- [Olsen, 1989]. Olsen, D. R. A Programming Language Basis for User Interface Management. In *proceedings of the CHI'89 conference on Human*

Factors in Computing Systems (Austin, Texas, April 30-May 4, 1989), ACM, 171-176.

[Olsen, 1990]. Olsen, D. Propositional Production Systems for Dialog Description. In *Proceedings of the ACM CHI'90 Conference on Human Factors in Computing Systems* (April 1990), 57-63.

[Payne, 1984]. Payne, S. Task-action grammars. In *In Human-Computer Interaction - IFIP INTERACT'84, Volume 1*, Elsevier, 1984, 139-144.

[Perls, 1969]. Fritz Perls. The Gestalt approach. 1969.

[Petrie et al, 1996]. Petrie, H., Morley, S., McNally, P., Graziani, P., Stephanidis, C., Savidis, A., Majoe, D. An interface to hypermedia systems for blind people. In the proceedings of the *ACM Hypertext'96* (demonstration).

[RACE, 1994]. Research and technology development in advanced communications technologies in Europe. RACE 1994. Commission of the European Commission, DG XIII, EUR-OP, Luxembourg, 1994.

[Reisner, 1981]. Reisner, P. Formal grammar and human factors design of an interactive graphics system. *IEEE Trans. Soft. Eng. SE-7,2* (March 1981), 229-240.

[RJCooper, 1997]. <http://www.rjcooper.com/page7.htm>.

[Saldarini, 1989]. Saldarini, R. Analysis and Design of Business Information Systems. Section on *Structured Systems Analysis*, MacMillan Publishing, New York, 1989, 22-23.

- [Savidis, 1994]. Savidis, A. The Concept of Dual Interfaces and the Design of a UIMS-Based Development Framework. *ICS-FORTH Technical Report*, February 1994, FORTH-ICS/TR-118.
- [Savidis et al, 1995a]. Savidis, A., Stephanidis, C. Developing Dual Interfaces for Integrating Blind and Sighted Users: the HOMER UIMS. In proceedings of the ACM CHI'95 conference in Human Factors in Computing Systems, Denver, Colorado, May 7-11, 106-113.
- [Savidis et al, 1995b]. Savidis, A., Stephanidis, C. Building non-visual interaction through the development of the Rooms metaphor. In companion of the *CHI'95 conference in Human Factors in Computing Systems*, Denver, Colorado, May 7-11, 244-245.
- [Savidis et al, 1995c]. Savidis, A., Stephanidis, C Integrating the visual and non-visual worlds: developing dual user interfaces. In *proceedings of the RESNA'95 conference* (June 9-14), Vancouver, Canada, RESNA Press, 1995, 458-460.
- [Savidis et al, 1996]. Savidis, A., Stephanidis, C., Korte, A., Crispian, K., Fellbaum, K. A Generic Direct-Manipulation 3D-Auditory Environment for Hierarchical Navigation in Non-visual Interaction. In proceedings of the *ACM ASSETS'96 conference*, Vancouver, Canada, April 11-12, 1996, 117-123.
- [Savidis et al, 1997a] Savidis, A., Stephanidis, C., Akoumianakis, D. Unifying Toolkit Programming Layers: a Multi-purpose Toolkit Integration Module. In *proceedings of the Eurographics DSV-IS'97 workshop in Design, Specification and Verification of Interactive Systems*, Granada, Spain (June 4-6). Harrison, M., Torres, J. (Eds). Springer-Verlag, Wien, 1997, 177-192.
- [Savidis et al, 1997b]. Savidis, A., Vernardos, G., Stephanidis, A. Embedding scanning techniques accessible to motor-impaired users in the WINDOWS object library. In, *Design of Computing Systems: Cognitive*

Considerations (21A). Salvendy, G., Smith, M., Koubek, R. (Eds). Elsevier, 1997, 429-432.

[Savidis et al, 1997c]. Savidis, A., Stergiou, A., Stephanidis, C. Generic containers for metaphor fusion in non-visual interaction: the HAWK interface toolkit. In *proceedings of the Interfaces 97 Conference, Session M2 - Devices for the Disabled*, May 1997, France, 194-196.

[Savidis et al, 1997d]. Savidis, A., Stephanidis, C. Unified manipulation of interaction objects: integration, augmentation, expansion and abstraction. In *proceedings of the 3rd ERCIM Workshop on User Interfaces for All*, November 3-4, INRIA Lorraine, 75-89.

[Savidis et al, 1997e]. Savidis, A., Paramythis, A., Akoumianakis, D., Stephanidis, C. Designing user-adapted interfaces: the unified design method for transformable interactions. In *proceedings of the ACM DIS'97 conference in Designing Interactive Systems*, Amsterdam, Netherlands, August 18-20, 323-334.

[Savidis et al, 1997f]. Savidis, A., Stephanidis, C. Agent classes for managing dialogue control specification complexity: a declarative language framework. In *Design of Computing Systems: Cognitive Considerations (21A)*. Salvendy, G., Smith, M., Koubek, R. (Eds). Elsevier, 1997,461-464.

[Savidis et al, 1997g]. Savidis, A., Akoumianakis, D., Stephanidis, C. Software Architectures for Transformable Interface Implementations: Building User-Adapted Interactions. In *Design of Computing Systems: Cognitive Considerations (21A)*. Salvendy, G., Smith, M., Koubek, R. (Eds). Elsevier, 1997,453-456.

- [Savidis et al, 1997h]. Savidis, A., Stergiou, A., Stephanidis, C. Metaphor Fusion in Non-visual Interaction. In *HCI International'97, Poster Sessions: Abridged Proceedings*, Elsevier, 1997, 61.
- [Savidis et al, 1997i]. Savidis, A., Petrie, H., McNally, P., Ahonen, J., Koskinnen, M., Stamatis, C. Internal report on the usability evaluation of the PIM toolkits. ACCESS Consortium, January 1997.
- [Savidis et al, 1998]. Savidis, A., Stephanidis, C. The HOMER UIMS for Dual Interface Development: a Step Beyond Adaptations of Visual Dialogues. Conditionally accepted for publication, *IwC Journal*, 1998.
- [SAW, 1992]. Switch Access to Windows 3. Reference Manual, ACE Centre, UK, 1992.
- [Short, 1997]. Component based development and object modeling. Texas Instruments Software, version 1.0, February 1997.
- [Singh et al, 1989]. Singh, G., Green, M. A high level User Interface Management System. In *proceedings of the CHI'89 conference on Human Factors in Computing Systems* (Austin, Texas, April 30-May 4, 1989), ACM, 133-138.
- [Stary, 1996]. Stary, C. Integrating workflow representations into User Interface design representations. In *Software Concepts and Tools*, Vol. 17, December 1996.
- [Stephanidis, 1995], Stephanidis, C. Towards User Interfaces for All: Some Critical Issues. Panel session on User Interfaces for All: Everybody, Everywhere, and Anytime. *6th International Conference on Human-Computer Interaction* (HCI International'95), Tokyo, Japan, July 9-14, Vol 1, 137-142.

[Stephanidis et al, 1995a]. Stephanidis, C., Savidis, A. Towards Multimedia Interfaces for All: a New Generation of Tools Supporting Integration of Design-time and Run-time Adaptivity Methods. In *ACM Multimedia'95 Workshop on Adaptive Technologies for People with Disabilities*, November 11, San Francisco, 1995.

[Stephanidis et al, 1997]. Stephanidis, C., Akoumianakis, D., Ziegler, J., Faehnrich, K-P. User Interface Accessibility: A retrospective of current standardisation efforts. In *Design of Computing Systems: Cognitive Considerations, 21A*, G. Salvendy, M. Smith and R. Koubek (Eds), Elsevier, 1997, 469-472.

[Stephanidis et al, 1997a]. Stephanidis, C. Savidis, A, Akoumianakis, D. Unified Interfaces Development: Tools for Constructing Accessible and Usable User Interfaces. All day tutorial #13, August 26, HCI International'97, San Francisco.

[Stephanidis et al, 1997b]. Stephanidis, C., Paramythis, A., Savidis, A., Sfyarakis, M., Stergiou, A., Leventis, A., Maou, N., Paparoulis, G., Karagiannidis, C. Developing Web Browsers Accessible to All: Supporting User-Adapted Interaction. In *proceedings of the 4th European Conference for the Advancement of Assistive Technology (AAATE '97)*, Thessaloniki, Greece, 29 September - 2 October 1997.

[Stephanidis et al, 1997c]. Stephanidis, C., Paramythis, A., Karagiannidis, C, Savidis, A. Supporting Interface Adaptation: the AVANTI Web-Browser. In *proceedings of the 3rd ERCIM Workshop on User Interfaces for All*, Obernai (France), November 3-4, 5-18.

[TAE Plus, 1998]. <http://www.cen.com/tae/>.

- [Ten Hagen, 1990]. Ten Hagen, P. Critique of the Seeheim Model. In *User Interface Management and Design*. Duce, D., Gomes, M., Hopgood, F. (Eds). Eurographic Seminars, Springer-Verlag, 1990, 3-6.
- [Thomas et al, 1995]. Thomas, M., Zahavi, R. *The Essential CORBA: Systems Integration Using Distributed Objects*, John Wiley & Sons, 1995.
- [TRACE Centre, 1995]. TRACE Research and Development Centre. *The Principles of Universal Design. Version 1.1 - 12/07/95*.
- [Tsichritzis, 1997]. Tsichritzis, D. How to Surf the Technology Waves We Created - the Human Connection. In *CACM Journal Vol 40 (2)*, 49-54, 1997.
- [UIMS, 1992]. The UIMS Developers Workshop. A meta-model for the run-time architecture of an Interactive System. *SIGCHI Bull 24, 1* (January 1992), 32-37.
- [Vergara, 1994]. Vergara, H. PROTUM - A Prolog based tool for user modelling. Bericht Nr. 55/94 (WIS-Memo 10), University of Konstanz, Germany, 1994.
- [Voyager, 1997]. VOYAGER. Agent-enhanced Distributed Computing for Java™. User Guide version 1.0, Beta 2.1, ObjectSpace, Inc.
- [Wellner, 1989]. Wellner, P. Statemaster: A UIMS based on Statecharts for Prototyping and Target Implementation. In *Proceedings of the AC CHI'89 Conference on Human Factors in Computing Systems* (Seattle, Wash., April 1-5, 1990), 177-182.
- [WinSCAN, 1997]. WinSCAN Version 2.0. <http://www.acsw.com/ws1.html>.
- [WiVik, 1997]. <http://www.prentrom.com/access/wivik.html>.

- [Wise et al, 1995] Wise, G. B., Glinert, E. P. Metawidgets for multimodal applications. In *proceedings of the RESNA'95 conference*, Vancouver, Canada, June 9-14, 455-457.
- [Wolber, 1996]. Wolber, D. Pavlov: Programming by Stimulus-Response Demonstration. In proceedings of the CHI'96 conference on Human Factors in Computing Systems, Vancouver, British Columbia, Canada (April 13-18), ACM, 1996, 252-259.
- [X Consortium, 1994]. FRESCOTM Sample Implementation Reference Manual. X Consortium Working Group Draft, Version 0.7, April 9, 1994.
- [Zanden et al, 1990]. Zanden, B. V., Myers, B. A. Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces. In *Proceedings of the CHI'89 Conference on Human Factors in Computing Systems* (Seattle, Wash., April 1-5, 1990) ACM, New York, 1990, 27-34.
- [Zhao, 1993]. Zhao, R. Incremental recognition in gesture-based and syntax-directed diagram editors. In *Proceedings of the INTERCHI'93 Conference on Human Factors in Computing Systems* (Netherlands, Amsterdam, April 24-29, 1993), ACM, New York, 1993, 95-100.

ANNEX - AN EPILOGUE

Users Acting - How Conscious their Decisions Might be ?

...a reference to Plato

Modern existentialism echoes Plato in its acceptance of a rational level of reality combined with its emphasis on the limits of reasoning and logical thinking. At the core of the duality of existence in the rational and mystical is the issue of consciousness and free will. Plato expressed the profound paradox inherent in the concept of consciousness and man's ability to freely choose. When transferring the findings and questions of Plato in all aspects of man's life, it is questionable whether people which all seem to behave differently, being conscious of their acts and decision making, do perform in a way which, being regulated by the natural laws from their birth, allows freedom of selection. In other words, our decisions at any point rely upon a mechanistic processing of knowledge and stimulus, triggering the evolution of our personal character, which all have as a starting point the first reaction of our biological recordings with the initial physical stimulus.

Interacting or Interworking with the Computer ?

... a reference to Lorenz

In real life, there is not always a clear relationship between external phenomena which occur sequentially, since causes and effects may be also internal within an living environmental system. This is the primary reason that the notion of feedback interpretation in its primitive form is related to instinctual recordings. For instance, the systematic application of *post hoc*, i.e. after a phenomenon, hence, because of the phenomenon, while it is considered in broader philosophical terms as totally irrational, since it is based on the identification of a cause for a phenomenon

something which simply preceded that phenomenon, it has been proved from many experiments to be an instinctual process.

In the context of Human-Computer interaction, direct manipulation guidelines emphasise the unique mapping of actions to corresponding effects, primarily because, in that specific school of thought, interfaces should not encompass internal independent sources of actions, but merely respond to user actions. Hence, it may be argued that the nature of our instinctual mechanisms, recorded in our biological inheritance system, complies more to the notion of a fully controlled interface, in which post hoc feedback is only supported (i.e. reactions), as opposed to the concept of a computer “working” together with the human-user, thus, performing independent actions and causing non post-hoc feedback effects.

Is Interaction with the External World just an Illusion ?

... a reference to Descartes

Descartes pushed rational scepticism to its limits. Acknowledging that the existence of other people and even our own bodies may be illusions, he concluded that we cannot doubt the existence of our own thought and hence his famous conclusion “I think, therefore I am”. Descartes implicitly associates such an illusion with man’s mind, being mainly a product of our own thinking, or being the overallness of thought itself. Our interaction with what is being conceived as being the external world is an illusive experience, differentiating the fundamental existential question not for ourselves, but for the rest of the universe.

On the Understanding of Things Towards the Quest of Truth

... a reference to Goethe

“All things are more simple than anyone could ever imagine, and in the means time more complex than anyone could ever understand. The largest difficulties exist there that we are not looking for them.”, excerpts from *Maximen und Reflexionen*.

The Art of Creation - Engineering, Inspiration or Both ?

...a reference to Leonardo da Vinci and Leone Battista Alberti

“You learn the arts by firstly studying the method and then you conquer them through the technique”. This old saying by Leone Battista Alberti emphasises the importance of a strategy in finding a way through arts (learning), as well as the need for excellence in implementing artefacts, in order to draw your own path (innovating). For Leonardo da Vinci, the design process as such seems to deserve longer time, being a more intellectual demanding phase, than implementation itself: “genius men make outstanding creations when they work [with their hands] less, because they have time to think and perfect with their own hands what they have conceived with their mind”.

On the Perfection of Design

... a reference to Michael Angelo

“You have reached perfection of design not when you have nothing more to add, but nothing more to remove”. This is a famous saying in the context of software engineering and design patterns, relying upon the common wisdom that “simplicity leads to perfection”. In its finest application we position the excellent technique of Michael Angelo in creating sculptures with a live form: “...he used to place in front of him the cold stone, and look at it, seeing a form to emerge and liberate from its prison...then, he would apply his outstanding technique to remove those pieces [of stone] which were not part of the envisioned form”.

Interaction and Children - “the Naïve Experts”

... a reference...to the future

“I...chose six children ages seven to nine who had been working with computers for several years...I told each child (one at a time) that I would ask them several questions and that there was no right or wrong answer, I just wanted their opinion. The questions were, Can a computer remember ? Does a computer learn ? Do computers think ? Do computers have feelings ? Do you like computers ? Do computers like you ? To the first two questions each child answered in the affirmative: computers do remember, and they do learn. The third question required a few moments of reflection, and all but two of the children concluded that yes, computers do think.

Apparently, one important clue to computers’ thinking ability for the children was the fact that when the children ask a computer to do something, it sometimes answers right away and sometimes there is a delay while the computer apparently *thinks* about the task for a while before responding. This, the children felt, was very similar to the way that they respond to questions. The fourth question - Do computers have feelings ? - not only was unanimously answered in the negative; it generally elicited laughter as if I asked, Do elephants fly ? Laughter, according to Freud, sometimes results from the juxtaposition of two concepts that are not supposed to go together, which may result from either the two concepts’ never having ever been linked before or a social taboo. Possibly both reasons caused laughter in this case. On the fifth question, all of the children answered affirmatively that they like computers. All of the children thought that the last question was silly, that computers do not have likes and dislikes”, from [Kurzweil, 1992].

“The computer that my uncle has is better than the one we have at school. It has colours, it is smaller, it can speak, it does “bips”, and it even has a mouse!”

Dimitris, 7 years old, 1996, Greece