



National Library  
of Canada

Acquisitions and  
Bibliographic Services Branch

395 Wellington Street  
Ottawa, Ontario  
K1A 0N4

Bibliothèque nationale  
du Canada

Direction des acquisitions et  
des services bibliographiques

395, rue Wellington  
Ottawa (Ontario)  
K1A 0N4

Your file    Votre référence

Our file    Notre référence

## NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

## AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

**AN APPLICATION-ORIENTED USER INTERFACE MODEL  
AND DEVELOPMENT SYSTEM**

BY  
Haiying Wang



A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment  
of the requirements for the degree of Doctor of Philosophy

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta  
Spring, 1994



National Library  
of Canada

Acquisitions and  
Bibliographic Services Branch

395 Wellington Street  
Ottawa, Ontario  
K1A 0N4

Bibliothèque nationale  
du Canada

Direction des acquisitions et  
des services bibliographiques

395, rue Wellington  
Ottawa (Ontario)  
K1A 0N4

Your file    Votre référence

Our file    Notre référence

**The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.**

**L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.**

**The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.**

**L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.**

ISBN 0-612-11407-4

**Canada**

**UNIVERSITY OF ALBERTA**

**RELEASE FORM**

**NAME OF AUTHOR:** Haiying Wang

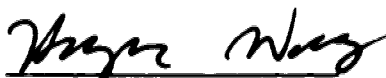
**TITLE OF THESIS:** An Application-Oriented User Interface Model and Development System

**DEGREE:** Doctor of Philosophy

**YEAR THIS DEGREE GRANTED:** Spring, 1994

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

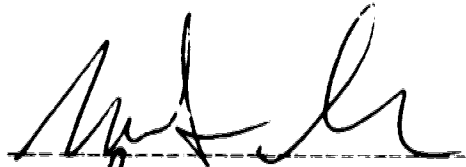
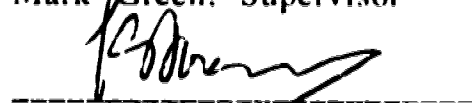
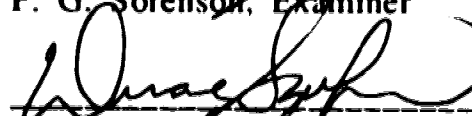


A handwritten signature in black ink, appearing to read 'Haiying Wang', is written over a horizontal line.

1177 Amarillo Avenue, #1  
Palo Alto, CA 94303  
U.S.A.

# UNIVERSITY OF ALBERTA

## FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled An Application oriented User Interface Model and Development System submitted by Haiying Wang in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

  
\_\_\_\_\_  
Mark Green, Supervisor  
\_\_\_\_\_  
P. G. Sorenson, Examiner  
\_\_\_\_\_  
D. Szafron, Examiner  
\_\_\_\_\_  
N. Durdle, Examiner  
\_\_\_\_\_  
A. Basu, Examiner  
\_\_\_\_\_  
D. R. Olsen, Examiner

U.M. Maydell, Committee Chairman

November 16, 1993

# Abstract

Existing user interface models in UIMSs such as the **Seeheim** model emphasize the abstraction of dialogue from the application computation. They support top-down architectures by providing a complete set of top level components and leave designers with the task of configuring or instantiating them. These models tend to restrict the design of various applications. Component models in existing interface toolkits, such as the **MVC** (Model-View-Controller) model, focus on a component abstraction for an individual interface object. They provide programming abstractions, but not an architecture.

This thesis explores a particular model — an application-oriented architectural model for user interfaces that provides abstractions for constructing various application-specific user interface structures and leads to an effective implementation of a user interface development tool.

The major results of this research are the following:

- A new user interface architectural model was developed. The model identifies the components the user interface designer needs when describing a user interface's structure and provides mechanisms that allow the designer to compose these components to meet the application needs. The model provides a set of middle-level components, compared with the top-level components in the **Seeheim** model and low-level blocks in the **MVC** model, together with a composition means that supports the construction of a wide variety of application-specific structures for interactive systems.
- Two new user support frameworks: undo and customization were developed on top of the above architectural model. They support the construction of effective support facilities in a user interface.
- A non-trivial user interface development tool, called the *User Interface Structure Design Tool (UISDT)*, was developed. UISDT, a tool based on the architectural model,

allows the designer to produce a user interface using a graphical editing metaphor. UISDT supports the designer in defining an object model of his/her application by providing abstractions and in implementing the application by producing a prototype from the object model.

# Acknowledgments

I am deeply indebted to my advisor, Professor Mark W. Green, for his guidance, encouragement, and friendship throughout the course of my study at University of Alberta. He made me believe I could finish and helped me do it with his insight and technical talents, and I am grateful. I also thank Professors Duane Szafron and Paul Sorenson for serving diligently on my thesis committee and their valuable comments and suggestions. Thanks also to Professors D. R. Olsen Jr., N. Durdle, and A. Basu for serving on my reading committee and their comments.

I would also like to thank the many other people who have had an impact on this work from a technical standpoint. Among those are my fellow students Jiandong Liang, Chris Shaw, Gurminder Singh, and Chun Ye, with whom I had countless discussions both relevant and irrelevant topics. Many ideas in this work are the direct result of these interactions, and I am sure things would have turned out much differently without their help. Jiandong and Chun helped me to experiment with early version of my UISDT system and have provided helpful feedback. I also have received valuable technical assistance from Alynn Klassen, former manager of operations at Computing Science department. Thanks also to Dr. Mark Linton and his InterViews group at Stanford University for their constructive comments when I was joining their regular group meetings.

Then there are those from whom I have benefited in other ways. I have had the pleasure of working alongside the particular entertaining and simulating people that make up the Graphics group, including Hanqiu Sun, Yunqi Sun, and Robert Lake. Dekang Lin, Zhiyong Liu and Yong Hu deserve special mention as close friends and frequent lunch companions. Wenping Wang, Anwei Liu and Dan were great office mates.

Finally, I could never even have started this undertaking without the love and support of my wife Ning Xu, my little daughter Regina, and my parents. Mum and Dad, thanks for instilling in me the thirst for knowledge and the strength to persevere. Regina, thanks



for making our life a great fun. And Ning, thanks for all for your love, companionship, and friendship. I dedicate this work to you.

# Contents

<b>Abstract</b>	<b>2</b>
<b>Acknowledgments</b>	<b>4</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problems . . . . .	2
1.2 Terminology . . . . .	4
1.3 The Thesis . . . . .	4
1.4 Goals . . . . .	6
1.5 Results and Contributions . . . . .	7
<b>2 Related Work</b>	<b>8</b>
2.1 User Interface Management Systems . . . . .	8
2.1.1 User Interface Models in UIMSs . . . . .	8
2.1.2 User Interface Construction in UIMSs . . . . .	14
2.2 User Interface Builders . . . . .	17
2.2.1 Component-Oriented UI Models . . . . .	17
2.2.2 User Interface Builders . . . . .	21
2.2.3 Summary . . . . .	22
2.3 User Support Facilities . . . . .	22
2.3.1 Undo . . . . .	22
2.3.2 Customization . . . . .	24
<b>3 A User Interface Architectural Model</b>	<b>26</b>
3.1 The Application-Oriented Model . . . . .	27
3.2 Domain Mapping Approach . . . . .	29

3.3	The Architectural Model . . . . .	31
3.3.1	Basic Abstractions . . . . .	31
3.3.2	User Interface Objects . . . . .	33
3.3.3	User Interface Object Organization — UI Structures . . . . .	38
3.3.4	Prototype . . . . .	43
3.4	Designing An Interface with the Model . . . . .	49
3.5	Summary . . . . .	49
<b>4</b>	<b>User Support Facilities</b>	<b>51</b>
4.1	An Event-Object Recovery Framework . . . . .	51
4.1.1	The Problem . . . . .	52
4.1.2	The Event-Object Model . . . . .	53
4.1.3	The Recovery Framework . . . . .	55
4.1.4	Performance of the Framework . . . . .	64
4.1.5	Summary . . . . .	66
4.2	User Customization Facility . . . . .	66
4.2.1	Domain Independent Approach . . . . .	68
4.2.2	The Architecture . . . . .	69
4.2.3	Basic Abstractions . . . . .	71
4.2.4	Prototype . . . . .	75
4.2.5	Experience . . . . .	77
4.2.6	Summary . . . . .	78
<b>5</b>	<b>UISDT — A User Interface Design Tool</b>	<b>80</b>
5.1	UISDT in Action . . . . .	81
5.2	Architecture of UISDT . . . . .	84
5.3	Design Process . . . . .	87
5.3.1	Component Definition . . . . .	88
5.3.2	Relationship Specification . . . . .	91
5.3.3	User Support Facilities . . . . .	93
5.4	Code Generation . . . . .	94
5.5	Discussion . . . . .	95

<b>6</b>	<b>Experimental Applications</b>	<b>97</b>
6.1	Performance Monitor . . . . .	97
6.1.1	POAT's Functionality . . . . .	98
6.1.2	POAT's Architecture . . . . .	99
6.2	Chinese Chess Program . . . . .	101
6.3	Enterprise User Interface . . . . .	104
6.3.1	Enterprise Concepts . . . . .	105
6.3.2	User Interface Functionality . . . . .	106
6.3.3	User Interface Implementation . . . . .	109
6.4	Experience With UISDT . . . . .	110
6.4.1	Generality . . . . .	110
6.4.2	Productivity . . . . .	112
<b>7</b>	<b>Conclusion</b>	<b>114</b>
7.1	Summary of Work and Contributions . . . . .	114
7.2	Future Work . . . . .	116
7.2.1	Productivity Measurement . . . . .	116
7.2.2	New Features in the Model . . . . .	116
7.2.3	Component Refinement . . . . .	117
7.2.4	UISDT Extension . . . . .	118
7.2.5	Powerful User Support Facilities . . . . .	118
	<b>Bibliography</b>	<b>120</b>

# List of Tables

1	AAD object protocol . . . . .	46
2	IV object protocol . . . . .	48
3	Recovery information protocol . . . . .	58
4	Recovery object protocol . . . . .	59
5	Customization object protocol . . . . .	75
6	Modeling component protocol . . . . .	76

# List of Figures

1	Seelheim UI model . . . . .	9
2	The Model-View-Controller model . . . . .	18
3	The abstract architecture of an interactive system . . . . .	30
4	The logical components of an interactive systems . . . . .	32
5	User interface object model . . . . .	36
6	Hierarchical structure of the model . . . . .	38
7	High-level decomposition of the chess program . . . . .	41
8	Chess program structure: middle level abstraction . . . . .	43
9	Piece object with two views: component abstractions . . . . .	44
10	User interface design pyramid . . . . .	50
11	The Components of the undo framework . . . . .	56
12	Communication during recording and recovering process . . . . .	60
13	The examples of the recovery facility structure . . . . .	63
14	Structure of the customization framework . . . . .	71
15	Communication between objects during customization . . . . .	74
16	Building application model . . . . .	83
17	Building view model . . . . .	85
18	Building user interface structure . . . . .	86
19	Object-oriented software development circle . . . . .	88
20	Define Component . . . . .	90
21	POAT system prototype and structure design in UISDT . . . . .	99
22	Chinese chess program . . . . .	102
23	Chinese chess program structure in UISDT . . . . .	103
24	Enterprise at work . . . . .	107
25	Enterprise user interface structure in UISDT . . . . .	111

# Chapter 1

## Introduction

Graphical user interfaces (GUI) for workstation applications are difficult to design and expensive to build. The creation of a user interface is currently a high percentage of the cost of application software. This percentage will increase as users demand higher-quality interfaces and more powerful interaction devices become available. One way to reduce the difficulty and the cost of user interface creation is to provide designers with better development tools. This research investigates the architectural issues of UI construction: the selection of components from which they are composed, the interactions among components, and the composition of interacting components. The UI structure is a very important part of a UI construction, yet structural design is the least supported part of current UI development tools. This dissertation describes the design and implementation of UISDT — an object-oriented user interface structure development tool, that allows UI designers to create a UI by building up its structure. UISDT embeds an application-oriented user interface architectural model whose abstractions make it easier for designers to build application-specific UI structures. These abstractions are middle-level components targeted specifically towards object-oriented GUIs with powerful user support features such as undo and customization. UISDT is a design tool that focuses on the architectural issues and supports the rapid prototyping of a UI. UISDT demonstrates that the effort of creating a graphical user interface can be greatly reduced if these middle-level components and their composition mechanisms are carefully designed and implemented.

## 1.1 Problems

UI research has provided extensive theory and techniques for the creation of interaction styles and there is no longer any reason for interfaces to be flawed by such problems as inadequate menu selection techniques and inconsistent commands. However, the use of the best available interaction techniques does not assure the production of a good UI. GUI development is no longer dependent only on advances in technology to provide increased functionality. Success in UI creation comes when a UI properly addresses the semantics of its users' tasks and abilities. Usability of a GUI is increasingly becoming a design issue, how to define UI components and how to organize them. One way to incorporate the semantics of the task and the domain into the UI is to let designers specify the application-specific structures of a UI, since much of the quality of an interactive system can be traced to the software structure which underlies it. There are clear relationships between the quality of an interactive system and not only the software components of that system, but also the embracing structure of these components. That is, architectural support is an important way to create a good UI and the key design issue is an architectural one. A good architectural model reduces the possibility of inflexible design and leaves designers free to concentrate on the important details of interfaces. A bad architectural model forces the designers' hand, making it difficult to make some design decisions.

Two major approaches for user interface development are *user interface management systems* (UIMS) and *user interface toolkits*. The UIMS approach provides an environment for producing quality interfaces faster and easier. UIMSs are usually based on a linguistic paradigm and emphasize the separation of the interaction dialogue (syntactic level) from the application computation (semantic level). UIMSs factor out user-application dialogue by providing a domain of large-grain dialogue traces, along with some high-level notation to describe them. Most existing UI architectures in UIMSs bear some resemblance to the **Seeheim** model [Gre85a] that provides a complete set of top level components and supports a top-down design method. They make the top level components application-independent by cutting across application dependent categories. They allow UI designers to concentrate on dialogue design and leave the details of the implementation to the underlying system. However, in UIMSs it is usually difficult for a designer to create new components, though the designer can easily identify subcomponents of the given top level components. The problem with UIMSs is their traditional premise: that UI software can be separate from



the application. The confusion is that this split works on a small scale, on particular applications, and at a conceptual level, but becomes hopelessly complicated and difficult when applied on a large scale, and a wide range of applications.

The toolkit approach provides a programming abstraction for building interfaces with a kit of well designed components that are general enough to cover the interaction requirements of most applications. GUIs are typically object-oriented because direct manipulation involves the user associating appearance and behavior with an object on the screen. Major toolkits are based on the object-oriented paradigm and component abstraction encapsulates data, computation, input and output of each individual interactive object. Toolkits factor out the interactive functionality of applications by providing a domain of generic components. Rather than limiting UI designs to combinations of a limited set of high-level components, designers can have a relatively large selection of low-level components and in addition, toolkits vary considerably in the way they are coupled to the rest of an application. There are several models, such as Model-View-Controller (MVC) [GR83] and **active value** [HH88, Mye88], for facilitating the construction of object-oriented UI objects. They provide flexibility in the way that interactive components are coupled to the rest of an application and support the bottom-up design method. However, they provide only components and no architecture support. They only provide implementation support for individual UI objects and force the designers to specify the UI in great detail. The problem with the toolkits and their UI builders is the assumption that interactive systems are frequently built “bottom-up”, however, applications are built “top-down”. The result is that toolkits are overwhelming in complexity and underwhelming in functionality to designers.

User support means a class of facilities that assist the user’s ongoing interactions and enhance the usability of UIs. User support includes online help, user recovery, and user modeling. User support has been considered one of the important issues in constructing good UIs. However, effective support facilities need direct access to the semantics of the interactions, the knowledge about the context of actions and what meaning a particular sequence of actions might have, as well as knowledge about the context of interface objects that might be inferred from the fact that a user is manipulating a certain collection of objects. This requires that the support facilities penetrate into the internal structure of the system and the supporting concepts pervade the system. Previous work has been *ad hoc* and highly application specific.

The remainder of this chapter defines terminology that is used throughout this dissertation, states the thesis, describes the goals, and presents results and contributions of this research.

## 1.2 Terminology

The term “application” has several meanings. We define an application to be the total system that is developed for its end users. An interactive application consists of application domain software and UI software. The application domain, or simply the domain, is the field of interest of, and reason for, the application.

The term *user interface management system* (UIMS) is widely used both to refer to an architecture emphasizing separation between the UI and the application semantics and to refer to the tools used to specify and execute the UI. We will use the term to designate integrated environments that are built on top of window management systems and programming facilities and that helps an interface designer to create and manage many aspects of UIs, such as design, prototyping, execution, and maintenance.

*Widgets*, also called interaction techniques, are ways to use input devices to enter information into the computer and are application independent high-level functions built on top of window systems, such as buttons, dialog box, etc. A UI *toolkit* is a collection of interaction object classes (widgets) employing specific interaction media with associated management capabilities (e.g., Motif from OSF, OpenLook from Sun, and MFC from Microsoft).

A UI *builder* is a system that lets the UI designer arrange UI elements into a prototype of the final interface. It allows UI toolkit components to be assembled on the screen without writing any code.

There are many users within the domain of UI. We refer to an end user who runs the final interactive system as the *user*, while the tool users who set up the overall architecture of the interactive system, create the software structure, or structure the software are called *designers*.

## 1.3 The Thesis

Creating a UI using a UIMS is like constructing a building with a set of large-grain frames, such as wall and roof frames. We are able to construct a particular building by instantiating

the frames with different appearances and functions. These frames are easy to use when we only need to construct a particular kind of building, but their value is limited if we are trying to create different kinds of buildings. UIs are more dynamic than buildings in terms of structures and we usually cannot immediately decompose an interactive systems into a set of fixed top-level components. Instead, higher levels of the final software structure will reflect a specific design, and good designs show that no two applications have the same structure.

Constructing a UI with a toolkit is like creating a building with a set of primitive construction elements, such as bricks, and wood. The simple structure of these components places fewer restrictions on their use, and therefore, different kinds of buildings can be composed from these components. The complexity introduced by a large number of components is countered by the simplicity of each component. Construction can be relatively easy to carry out if builders know the frame of the building and understand what works and doesn't work in the world of components. Constructing a UI is more complex than constructing a building. First, components in a toolkit are much more complicated than the bricks and wood, and designers usually do not have the understanding necessary to make appropriate use of toolkit components. Second, and more importantly, a software structure is a collection of development resources combined in a particular way. Combining components in UIs is more difficult than putting bricks together. Various toolkits provide various composition techniques and different components in a toolkit require different fitting methods.

A better architectural model, or a better set of UI building blocks, needs middle-level components. These components are lower than components in a UIMS in the sense that they do not define particular UI structures, but provide basic components together with composition means that can be combined in a prescribed way to form more complex components until the top levels of the applications are reached. They are higher than components in toolkits in the sense that they support structure construction by encapsulating certain system properties and provide abstractions that shield designers from extraneous detail of the components' composition mechanism and behavior implementation.

The above discussion brings out the thesis of this dissertation which has the following parts:

- a) Much of the information needed by a UI from an application's semantic side could be categorized into a relatively small number of abstract queries that work well across

a wide range of UI styles and application domains. Common protocols can be defined between interactive objects and application objects for these queries so that the interactive objects can access the application semantics.

- b) A UI architectural model need not support the construction of all other higher level components other than basic interactive and application components. Instead, it should provide the means — composition patterns, for combining these components to build up application-specific components or integrating user support components.
- c) An object-oriented GUI composed from these components is flexible and significantly easier to design than one designed with traditional design approaches.
- d) Such application-specific architectures can be implemented through the re-use of UI toolkits with substantially less effort than with other techniques.
- e) User support features of a UI are system properties, not just the properties of specific objects and are related to overall software structure. They should and can be addressed at the structure design stage.

## 1.4 Goals

To prove the above thesis, I set the following goals:

1. Develop a new UI architectural model that describes ways of creating interfaces from components that standardize their communication protocols, and each of these components supports specific aspects of a UI behavior.
2. Design new user support frameworks, undo and user customization, that provide application-independent approaches for constructing user support facilities by embedding basic mechanisms into the UI architectural model.
3. Build a prototyping system supporting the creation of UIs based on the above UI architectural model.
4. Demonstrate the practicality of this approach by building applications.

This research consists of three parts. First, the investigation of a user interface architecture model that supplies a set of basic components with composition mechanisms for

combining them. Second, two new user support frameworks: undo and customization. Finally, the design and implementation of a UI structure development system (UISDT) that provides the environment for connecting various parts of the architectural model and assist UI designers to design and implement object-oriented GUIs easily and quickly. Success of this research is measured in terms of how well UISDT supports UI development and how easy it is to use compared with existing UIMSs and conventional programming with UI toolkits.

## **1.5 Results and Contributions**

This dissertation contributes a novel approach for creating graphical UIs by implementing UISDT, a system that supports GUI design, implementation, and refinement. By identifying and characterizing a class of middle-level components and their relationships in applications, we are able to design and build a general software tool for supporting their development. The new UI architectural model employed in UISDT simplifies the design of GUIs, while the prototype implementation of UISDT simplifies their realization.

UISDT demonstrates that the UI architecture is a very important design issue and the proposed architecture shows a powerful and practical way to create a UI. The dissertation has demonstrated the viability of the architectural model and UISDT implementation by using UISDT to create GUIs in several different domains. Though these interfaces do not represent polished systems, they have proven themselves useful tools for their intended purposes.

This architectural model provides a non-trivial framework for building user support facilities. The abstractions provided by the primitive components of the facilities and the mechanisms that are nested into the UI structures make it significantly easier to incorporate them into UIs.

## Chapter 2

# Related Work

User interface development is currently a very active area of research. Work relevant to the project described here includes the following:

- user interface management systems,
- user interface builders,
- user support facilities.

In this chapter we describe the methodologies, the current developments, and systems that characterize these three areas of user interface development. We conclude by discussing the shortcomings of existing approaches and systems in achieving the goals of this research.

### 2.1 User Interface Management Systems

UIMS studies have been reviewed in several surveys which emphasize the techniques used to describe UIs [HH89, Gre86], tools used to construct UIs [Mye89, Lee90], and the generations of UIMSs. In this section, we review the UIMS studies by describing concepts, methodologies, techniques, and systems in the context of UI models and supporting environments.

#### 2.1.1 User Interface Models in UIMSs

The user interface model is an interpretation of a human-computer interaction domain. Models from cognitive psychology such as ACT [And83], GOMS [CMN83] and others [Ke89, Bod91] offer a way to understand the nature of the cognitive process and human activity

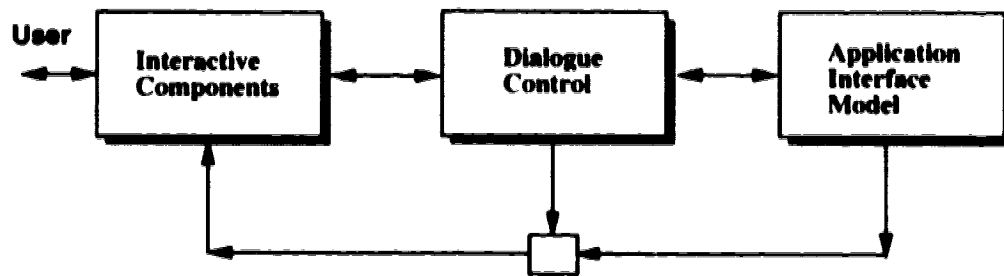


Figure 1: Seeheim UI model

involved in human-computer interaction. They are useful tools for thinking, but provide little help with the practical implementation of interactive systems. What we are interested in is an architectural model that describes the interfaces in terms of structure and provides designers with a framework for describing interactive software. The model defines how to separate the interactive component from computational component of an application at implementation time, while providing ways to combine these two components at run time.

The **Seeheim** model [Gre85a] is a general framework that can describe the UI structure of most interactive applications. The **Seeheim** model divides the UI into three logical components; the presentation component, the dialogue control component, and the application interface model (see figure 1), each of them has a different function in UIs. The presentation component is responsible for generating images on the display screen, accepting user input, and converting input data into the form required by the other components in the interface. The dialogue control component is responsible for managing the dialogue between the user and the application. It defines the structure of the dialogue and serves as a mediator between the user and the application. The application interface model is responsible for communications between the UI and the application. It is a representation of the functionality of the application from the UI's view and the functionality of the UI from the application's view.

The **Seeheim** model is an abstract model of the functionality of UIs that does not represent how a UI should be structured or implemented, many UIMSs' architectural models were based on or influenced by this logical model. Some UIMSs developed before the **Seeheim** model employed UI models that had similar ideas and can be considered as variants of the **Seeheim** model. We review the UIMSs in the terminology of the **Seeheim** model.

### Separation Between an Application and the UI

Separation of the interface from the computational component in design and implementation is crucial to easy modification of the interface by iterative refinement and has been the first principle of UI studies. Though different UIMSs may use different techniques to achieve the separation, their goals are the same, the separation is implemented by maintaining a strict division of responsibility between the UI and the application: the application did the work and the UI communicated with the user. As suggested in the **Seeheim** model, this separation is implemented by the application interface model that contains both the application view of the interface and the interface view of application. In COUSIN [HSL85], *slots* are used to separate the UI from application, a slot is defined for each piece of information the UI and application need to exchange and the applications access slots with a set of accessory routines. Open Dialogue [SRH85], FLAIR [WR82], RAPID/USE [Was85], and Sassafras [Hil86] use a similar approach by providing a special definition language to define the interface between the UI and the application. The interface view of an application is implemented as a set of application routines and the user actions are mapped to invocations of these application routines. In the UIMSs described in [Jac83, Ols84], nodes in state transition networks are used to specify the separation. A node describes where the user action would lead the system to (the system state) and what application procedure would be called in response to this action (the system response). In [Gre85b, SG91], the application interface model is expressed by a set of event handlers that accept user generated events, transform them to tokens, and pass tokens to application routines. A token is the smallest dialogue unit that has meaning to an application system. The event handlers separate the application from the interaction part.

These UIMSs' models achieved the separation by specifying a set of call-back application routines. They are called call-back routines because once the application starts, it gives control to the UI, which can then call the application back when appropriate. Call-back routines minimize the interaction between the application and the UI, which results in a low-bandwidth connection between the two, thus maximizing their independence. To overcome the limited bandwidth problem, the application interface model should include more application semantic information as is suggested by the **Seeheim** model [Gre85a]. In the Serpent UIMS [Bas88], data shared between the application and Serpent is used explicitly as the application interface model to achieve the separation. In the Higgens UIMS [HK88], a special semantic data model is used to define the interface that not only represents



the application data related with interaction, but also it represents and implements the application semantics associated with these data. In the knowledge based UIMS [FGKK88], frames are used to define the application interface which contains the information required by dialogue control for routing tokens to the appropriate place within the application and constraints on the use of the application routines.

### **The User Interface Description**

Given a UI model, how designers describe specific instances of a UI is the issue of description. A UIMS usually provides the interactive dialogue abstractions that allows the designer to concentrate on the structure of dialogues without concern for the interaction techniques involved in the dialogues. A UI specification method is a mechanism for designers to express and record their designs. Numerous techniques based on different interaction modeling approaches have been used to support the dialogue specification. One widely known and used approach is the language model that views the human-computer interaction from a linguistic viewpoint at conceptual, semantic, syntactic, and lexical levels [Mor81].

The techniques used to support this modeling approach include formal grammars [OD83, VPH83, SY88], state transition networks [Jac83, Ols84, SBK85, Jac86], and the dialogue transition model [HH87]. All of them can only describe sequential interactions — a conversation style. This style of interaction moves in a predictable manner from one part of the dialogue to the next and depicts request-response interactions. Grammars, finite state transition diagrams, and transition models which is a combination of the first two, provide formal ways to describe valid sequences of tokens. Thus, this style works well in describing most conventional text-based UIs and produces the entire UI controlled by a single dialogue.

Because of its linguistic nature, many results from formal language theory can be applied to this approach. This allows some kinds of analysis to be performed on a dialog, such as predicting some of its human factor characteristics [BF82]. However, it has been found that the linguistic model may not be the best way to deal with GUIs which usually use direct manipulation interaction style. This style provides a world model of some sort which the user can manipulate directly. The problems are that: a) although the linguistic model provides a useful framework for focusing on issues that occur within the semantic, syntactic, and lexical levels of a dialogue, it encourages the designer to view each of these levels in isolation, which is not compatible with interactivity where even simple lexical feedback sometimes requires a certain amount of semantic knowledge, and b) they have difficulty

ation semantics associated with these data. In the knowledge based UIMS [FGKK88], s are used to define the application interface which contains the information required dialogue control for routing tokens to the appropriate place within the application and constraints on the use of the application routines.

### **User Interface Description**

In a UI model, how designers describe specific instances of a UI is the issue of description. MS usually provides the interactive dialogue abstractions that allows the designer to concentrate on the structure of dialogues without concern for the interaction techniques used in the dialogues. A UI specification method is a mechanism for designers to express and record their designs. Numerous techniques based on different interaction modeling approaches have been used to support the dialogue specification. One widely known and popular approach is the language model that views the human-computer interaction from a linguistic viewpoint at conceptual, semantic, syntactic, and lexical levels [Mor81].

The techniques used to support this modeling approach include formal grammars [OD83, [3, SY88], state transition networks [Jac83, Ols84, SBK85, Jac86], and the dialogue transition model [HH87]. All of them can only describe sequential interactions — a conversational style. This style of interaction moves in a predictable manner from one part of a dialogue to the next and depicts request-response interactions. Grammars, finite state transition diagrams, and transition models which is a combination of the first two, provide different ways to describe valid sequences of tokens. Thus, this style works well in describing conventional text-based UIs and produces the entire UI controlled by a single dialogue. Because of its linguistic nature, many results from formal language theory can be applied to this approach. This allows some kinds of analysis to be performed on a dialog, such as predicting some of its human factor characteristics [BF82]. However, it has been found that the linguistic model may not be the best way to deal with GUIs which usually use a manipulation interaction style. This style provides a world model of some sort which the user can manipulate directly. The problems are that: a) although the linguistic model provides a useful framework for focusing on issues that occur within the semantic, syntactic, and lexical levels of a dialogue, it encourages the designer to view each of these levels in isolation, which is not compatible with interactivity where even simple lexical feedback sometimes requires a certain amount of semantic knowledge, and b) they have difficulty

the application data related with interaction, but also it represents and implements the application semantics associated with these data. In the knowledge based UIMS [FGKK88], frames are used to define the application interface which contains the information required by dialogue control for routing tokens to the appropriate place within the application and constraints on the use of the application routines.

### **The User Interface Description**

Given a UI model, how designers describe specific instances of a UI is the issue of description. A UIMS usually provides the interactive dialogue abstractions that allows the designer to concentrate on the structure of dialogues without concern for the interaction techniques involved in the dialogues. A UI specification method is a mechanism for designers to express and record their designs. Numerous techniques based on different interaction modeling approaches have been used to support the dialogue specification. One widely known and used approach is the language model that views the human-computer interaction from a linguistic viewpoint at conceptual, semantic, syntactic, and lexical levels [Mor81].

The techniques used to support this modeling approach include formal grammars [OD83, VPH83, SY88], state transition networks [Jac83, Ols84, SBK85, Jac86], and the dialogue transition model [HH87]. All of them can only describe sequential interactions — a conversation style. This style of interaction moves in a predictable manner from one part of the dialogue to the next and depicts request-response interactions. Grammars, finite state transition diagrams, and transition models which is a combination of the first two, provide formal ways to describe valid sequences of tokens. Thus, this style works well in describing most conventional text-based UIs and produces the entire UI controlled by a single dialogue.

Because of its linguistic nature, many results from formal language theory can be applied to this approach. This allows some kinds of analysis to be performed on a dialog, such as predicting some of its human factor characteristics [BF82]. However, it has been found that the linguistic model may not be the best way to deal with GUIs which usually use direct manipulation interaction style. This style provides a world model of some sort which the user can manipulate directly. The problems are that: a) although the linguistic model provides a useful framework for focusing on issues that occur within the semantic, syntactic, and lexical levels of a dialogue, it encourages the designer to view each of these levels in isolation, which is not compatible with interactivity where even simple lexical feedback sometimes requires a certain amount of semantic knowledge, and b) they have difficulty

domains and different requirements, the way of organizing domain concepts in the system and the way of representing them to the end user differ from application to application. That is, as the application domain models change, the resulting architectures change. For example, designing a UI for a database application requires an emphasis on the information being viewed and manipulated by the user. Organizing complex data structures may be of paramount concern for one application, while capabilities for mapping user actions into the behavior of the interface, and controlling the appearance and managing multiple views for the same information may outweigh other considerations in a graphical drawing editor application.

### **2.1.2 User Interface Construction in UIMSs**

A UIMS is a development environment comprising a set of tools for design, implementation, evaluation, and maintenance of UI software. The UI model employed in a UIMS serves as a framework for understanding and describing the elements of interfaces and for providing guidance for the UI construction, while the tools are doing the real job of UI development. There are a number of tools that have been created for various UIMSs and these tools are different from each other in terms of methodologies they support for UI design, implementation, and maintenance, and the ways they are used by designers. In this section, we review existing tools according to the tasks supported, such as design, implementation, and rapid prototyping, and the ways that designers use them.

Most of the tools in UIMSs can be classified as UI implementation tools in the sense that they are used in the implementation stage of UI construction. The designer specifies the UI in terms of interaction components and application routines using a specific form that the tool can understand. These tools help the designer to implement the UI design by choosing the pre-defined interaction techniques or generating the code to implement these components, producing the interface by combining the interaction code with application routines, and finally allowing the designer to modify the interface interactively. The specification is either in a declarative form using a special-purpose abstraction or a graphical form through interactive manipulation.

#### **Declarative Specification**

Existing UIMSs are based on the linguistic model, where the human-computer interaction is basically a dialogue. The dialogue is expressed as a language, the design of the UI is based

on the semantics, syntax, and lexicon for the language, and the tool generates the UI by producing an interpreter for the language. The specification of the dialogue can take many forms, including context-free grammars [OD83, VPH83, SY88], state transition networks [Jac83, Ols84, SBK85, Jac86], and menu trees [Kas82]. The advantage for this language based approach is that there exist well-developed techniques for the design and implement these tools. The tools based on these techniques can achieve the goal of automatic (or semi-automatic) construction of UIs. However, due to the inherent sequential character of language theory and its low level description techniques, this approach has limited capability for describing UI in terms of the styles of interaction, the structure of the UI control, and the level of specification. The interaction has to be command-based, the UI control is sequential, and the designer specifies the UI at the syntactic and lexical levels (command names, menu organization, and sequential rules).

The non-linguistic description approaches, such as the event model [Gre85b, Hil86, FB87, SRH85, SG91], dialogue cells [HH87], and templates [FGKK88, dBFM92b, Sze90], suggest a higher level of specification. In using the tools based on these approaches, the designer expresses his/her UI in terms of objects and actions in the UI, which is beyond the levels of syntax and lexicon of dialogue design. The specification form is usually based on higher level abstractions of interactions that have been implemented by pre-defined blocks in the UIMS. The tools accept the UI requirement, then choose the appropriate pre-defined blocks to implement them, and finally combine them with the application routines. We can see that the functionality of these tools covers not only the UI implementation, but also some part of UI design [FGKK88, SG91]. The problem with this approach is that usually the set of pre-defined blocks used by the tool is limited in the range of UIs they support and can only be used to generate simple UIs.

### **Interactive Graphical Specification**

Graphical interfaces, because they can help convey concepts in an application to the user through visual perception, have become very popular UIs. Due to the visual aspects, it is very difficult to describe a GUI in a declarative specification. However, the visual presentation of the UI is of primary importance in GUIs. To overcome this difficulty, many UI development tools, such as those in Blox [Rub82], Menulay [Bux83], Trillium [Hen86],

RAPID/USE [Was85], Peridot [Myc88], UofA\* [SG91], and GROW [Bar86], have been created using the idea of building the interface interactively and using a UI editor to build the UI, i.e. allowing the UI to be defined by placing objects on the screen using a mouse.

The interactive tools in Blox, Menulay, Trillium, and UofA\* let the designer place text, icons, and menu buttons on screen, draw interface layouts, and test the UI prototype which is exactly what the user will see when the application runs. These tools automatically generate the code to implement the interaction techniques and combine them with application routines. Each active item, such as menu items and icons, in the display is associated with an application routine supplied by the application developer and this routine is invoked when the user selects that item. The interaction techniques are usually fixed and pre-defined, but not sufficiently general to cover most of the common cases. Peridot goes one step further by allowing the designers to create the interaction techniques themselves. The designer manipulates primitives provided by Peridot to construct the more complicated interaction techniques he/she needs. However, there are several shortcomings with these tools: 1) they support the creation of a limited range of interfaces due to the small number and simple interaction techniques they can implement; 2) they promote a narrow connection between the interface to be created and the application as the UI models employed support the strict separation at the system level.

GROW is a comprehensive interactive environment for UI development, which is composed of a powerful UI structure editor — SOW and a powerful UI toolkit Impulse-86 [SDB86]. SOW lets the designer create graphical objects to be used in the UI and define their composite structure and graphical dependencies interactively. These objects are the extension of objects in Impulse-86 which manage user interaction. Once these UI objects have been created, GROW links the interface and the application that involves creating calls from the UI objects to the application routines. GROW is more powerful in that it allows the designer to define various UI objects for his/her UI interactively, but GROW still restricts the connection between the interface and the application call-back procedures.

### **Problems With User Interface Development Tools**

UIMSs have gained wide acceptance in the UI research community and the concepts and techniques for developing tools in UIMSs have drawn great attention from researchers which can be seen by the large number of papers on UIMSs that appear in conferences and

magazines, and the great deal of new tools developed by businesses. However, existing UIMSs' tools have several shortcomings:

**Enforce a rigid system decomposition** In UIMSs it is usually difficult for a designer to create new components, though the designer can easily identify subcomponents of the given top level components. The problem with UIMSs is their traditional premise: that UI software can be separate from the application. The confusion is that this split works on a small scale, particular applications, and at a conceptual level, but becomes hopelessly complicated and difficult when applied on a large scale, and a wide range of applications.

**Limit designers control over design decisions** Because of automation in most UIMSs, they provide UI designers with very little control over design decisions. However, generating good UI designs is intrinsically difficult. Principles of good UI design are not yet well specified, (at least not at implementation level), and cannot be used to automatically prune the relative large design space. It is hard to tell a UIMS about application-specific considerations that affect alternatives.

## 2.2 User Interface Builders

Few UI builders today are written from scratch. Most UI builders benefit from a toolkit based implementation. The toolkit approach provides a programming abstraction for building interfaces with a kit of well designed components that are general enough to cover the interactive requirements of most applications. Toolkits factor out some of the functionality of applications by providing a domain of generic components. Toolkits vary considerably in the way they are coupled to the rest of an application.

### 2.2.1 Component-Oriented UI Models

UI models for object-oriented UIs have been investigated for a long time under different contexts: such as the MVC in Smalltalk [GR83, KP88], the Subject-View(SV) in *InterViews* [LVC89] and in *Andrew* [And88] for toolkits, *GROW* [Bar86], *GWUIMS* [SHB86] and *Template* [Sze90] for UIMSs, *Active-Value* in [HH88, Mye88] for direct-manipulation interfaces, and *PAC* [Coo89, BC91], *Daemon* [Nir91], and *MacApp* [Sch87] for interactive systems. The most difficult and important part of these models is the mechanism that separates the interaction

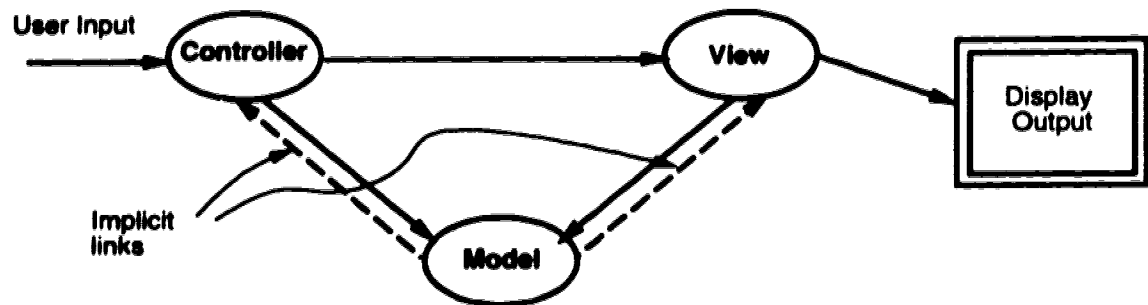


Figure 2: The Model-View-Controller model

part from the application at implementation time and combines the interaction part with the underlying data at run time.

The MVC model (shown on figure 2) has been widely used and influenced many other models in object-oriented paradigm. It is like the **Seeheim** model for non-object-oriented models that presents the logical components that appear in a UI object. In fact, MVC is like the **Seeheim** model in that it utilizes functional decomposition to develop the architecture of a UI object instead of an entire UI. The MVC paradigm has three components: model, view, and controller. The model in MVC represents the data structure of the application. The view deals with everything graphical; it requests data from its model, and displays the data. The view can contain not only the component needed for display, but can also contain sub-views and be contained within super-views. It is the views of interactive objects that make the composition of interactive objects possible. The controller contains the interface between its associated model and view, and the input devices. It also schedules interactions with other view-controller pairs. Each view-controller pair has one model, but each model can have several view-controller pairs. MVC is a highly coupled model in that the design and implementation of the three components are closely related. Although the MVC provides a compelling object-oriented division at the abstract-level, this coupling makes the independent design of the concrete interactive objects difficult and weakens the separation of interaction and semantic operations in the objects, which is very important in many aspects of UI design.

MoDE [Sha90] suggests the mode approach to overcome the shortcomings of MVC model. A mode, like a MVC object, contains three parts: appearance, interaction, and semantics. The mode approach decouples the connection between the components by defining a standard internal communication protocols between the three components and gives



the semantic component more control over the other two components. The mode approach supports reusability by encouraging the orthogonal design of the three components. However, many object-oriented UIs support direct-manipulation UIs that require the tight coupling between input and output of a manipulable object. Design and implementation of the appearance component and the interaction component of an object independently would increase the total number of objects in the interface and potentially decrease its efficiency.

The Subject-View (SV) paradigm in **InterViews** puts the functionality of MVC controller and view into one single object — the view that handles input and output, while the subject is similar to model in MVC. This consolidation reflects the tight coupling between input and output in direct-manipulation interface objects. The composition hierarchy of the objects is represented in the views of sub-objects. The views in SV are often involved in processing the semantics in addition to the input and output.

The **Active Value** approach is another well-known paradigm for combining interactive objects with the underlying data. The active values are similar to parameters to procedures. When a graphical object depends on an active value, a data constraint is automatically created, so the object will change immediately when the value is updated. In **GROW** [Bar86], an association table of graphical objects and its application-defined keys (active value) are used for the communication between the interactive components and application components. In **Peridot** [Mye88], active values are used as hooks to glue the interactive objects with the application part; they can be set by the application part at any time to update the graphics, application operations can be attached to the active value to pass information back to the application part when the active value changes. The disadvantage of the active value approach is that exposing the data structure to both interaction part and application part results in a tightly coupled structure and makes the orthogonal implementation of the two parts difficult.

**Daemon** [NMK91] is a coordination mechanism. A daemon attached to an object can monitor the object bound to a particular instance variable. Whenever a specific message is sent to the object, the daemon invokes a method of the object. Typical methods invoked by daemons maintain consistency between two objects. This mechanism can be used to connect an interactive part with its application part. The problem is how to specify a daemon which could be a mechanism similar to active value and how to implement a daemon mechanism without special support from the programming language.

In Humanoid [Sze90, SLN92], templates are used to connect interactive widgets with the abstract application data that specify how to map application data structures to the interactive widgets, how to break down complex data structures into substructures, and how to assign a widget to display each substructure. The difficulties are how to design the templates and how to reuse the predefined templates as they contain so much domain-specific knowledge about the connections.

The PAC model [Coo89] like MVC is a composite defined by its three components: Presentation, Abstraction, and Control. However, the Presentation defines both input and output behavior of the object, the Abstraction contains the functional core of the application semantics, and the Control maintains the consistency between the other two components. PAC has a different composition strategy: the hierarchy of a composite object is represented by the Controls of its components. The advantage of this composition approach is that it distributes semantic and syntactic processing at various levels of abstractions and suppresses boundaries between the application and the UI. PAC also uses Control to serve as an explicit bridge between the Abstraction and the Presentation. PAC's composition is based on the Controls which is recursively applicable except it is very difficult to apply this composition on graphical objects. Most composite objects in UIs are graphical objects which usually have their own state attributes. The state of a graphical composite should be defined as the combination of its components' state and its own. This leads to the composition depending on the Presentation and the Abstraction not on the Control.

GWUIMS is one of the early UIMSs that used an object-oriented paradigm. However, the object-oriented approach is used only as an implementation technique, while the design method is task-oriented and the UI model employed in GWUIMS is still the **Seehorn** model. GWUIMS focused on defining the boundaries of the lexical, syntactic, and semantic levels of interface language in terms of objects.

In general, studies investigating object-oriented UI models concentrated on the methods of combining the interactive part with the application part in UI objects which is no longer a simple set of call-back routines. The difficulty is to decide where to draw the line between these two parts and how to implement this decision. We know that an ideal separation is hard to define and even harder to achieve. The existing models have addressed this issue extensively, and have made great progress in both understanding and solving the issue, but none has adequately solved it. We can see the separation between the interactive behavior and application semantics is the key issue in a successful UI design and implementation.

There is no absolute measurement about the separation and therefore, there is no standard criterion for the best separation. The principle is to separate the interactive behavior and application part as much as possible in the UI development, but at the same time satisfying the application requirements and the end user demands. The basic requirement is to separate interactive components from computation components conceptually at the design stage, implement them independently, and integrate them at run time.

### **2.2.2 User Interface Builders**

UI builders [Mic90, VT91, Web89] are interactive graphical tools that allow UI toolkit components to be assembled on the screen without writing any code. Builders sit between UIMS and UI toolkits in that they allow some non-programmatic specification of UIs and do not have the strict separation of UI from the rest of application just as their underlying toolkits do not. Though the level of support offered can vary widely, a typical UI builder will allow a UI to be constructed using direct manipulation techniques, giving an essentially instantaneous WYSIWYG view of the UI under construction, and relieve the developer of at least some of the coding work which would otherwise be involved in building the UI. Many builders treat the external visual representation of the UI as the fundamental representation used by the designer. Components are added by dragging from a palette onto the wysiwyg view; the builder determines the internal structure of the UI the designer is creating, and then generates code to implement this UI.

UI builders promote visual UI design and development to minimize the need for conventional programming. Due to their connection with toolkits and their ease of use compared with toolkits, UI builders have become very popular UI development tools. Current commercial and research builders support mainly the construction of interaction objects and they differ in two major aspects: 1) the manner in which interaction objects are composed to create composite interaction objects and 2) the degree to which designers may change the appearance of interaction objects without having to reenter the relationships between interaction objects and the underlying application functions. These tools support UI specification on several levels: 1) most support the interactive layout of a UI, that is, absolute position of widgets by direct manipulation, 2) a few offer mechanisms for expressing spatial relationships between widgets, which define the resize semantics of an interface, and 3) some support hierarchical structuring of interaction objects to aid in the specification of complex layouts.

### 2.2.3 Summary

In summary, object-oriented models are more flexible than the function-oriented models in that they support multiple instances of components and architectures built up from lower level components: no single top-level composition is imposed. They provide flexibility in the way that interactive components are coupled to the rest of an application and support the bottom-up design method. However, toolkits' architecture models as well as UI builders suffer from the following problems in terms of UI structure construction:

**Force designers to handle too many design details** Working at the component level, builders or toolkits force designers to handle too much detail and designers are forced to make design commitments down to the level of individual widgets.

**Support only low-level component composition** Another problem with the toolkits and their UI builders is the assumption that interactive systems are frequently built "bottom-up", however, applications are built "top-down". The result is that toolkits are overwhelming in complexity and underwhelming in functionality to designers.

## 2.3 User Support Facilities

From a user's point of view a UI has task-oriented and support-oriented features. User support means a class of facilities that assist the user's ongoing interactions and enhance the usability of the UI. User support includes online help, user recovery, and user modeling. Their design and presentation have a major impact on the usability of an application. We discuss two types of user support facilities: undo and customization.

### 2.3.1 Undo

Undo/Redo features in interactive systems have been studied for a long time in several contexts. Interlisp [Tei75], COPE [Arc84], and PECAN [Rei84] are systems that have recovery facilities and US&R [Vit84] is a general undo/redo framework for interactive systems. Yang [Yan88] gives a comprehensive formal discussion of recovery features in interactive systems. While these recovery facilities seem to take different approaches to implementing recovery, all adopt the same idea. That is:

- a) use a history list to record all the issued primitive commands (commands which act on the objects in the systems) and their effects on the system,

- b) provide a set of meta commands, such as **undo**, **redo**, and **skip**, which operate on primitive commands in the recorded history list.

History lists can be implemented in several ways: a stack is used in InterLisp to store the commands, a file system is used to save the updated objects after executing a command in COPE, and a tree-like data structure organizes the executed primitive commands in US&R. All of these structures have the same purpose, recording recovery information for later recovery based on the recorded recovery information. The meta commands manipulate the history list to perform undo/redo operations. This mechanism works well for non-object-oriented interactive systems, where the interaction is based on a sequence of commands implemented by individual procedures. The procedures are the functional part of the system and the data structures represent the system state. The data structures are determined by the procedure structure and they tend to be globally accessible. This kind of system architecture provides a convenient mechanism for supporting domain-dependent and context-dependent recovery semantics at the system level. However, this traditional recovery approach is not suitable for object-oriented systems due to the structural differences between non-object-oriented software and object-oriented software.

There have been several studies on recovery facilities related to object-oriented methods. Rathke [Rat87] proposed a recovery mechanism using object-oriented techniques. The basic unit of recovery in his approach is the individual object. The recovery operations are included as methods in a recovery object. The state variables, which have been changed when operations are performed on the object, are collected within a slot of this recovery object. Another slot in this object records the transition functions and their corresponding inverse functions. The interaction history is a list of recovery objects. Though an object-oriented method is used to implement the history list, this framework is intended for systems designed using a non-object-oriented methodology. The recovery approach is purely command-oriented in the sense that each reversible system function has its own recovery object and data structures that are globally accessible. It would violate object structuring rules to use this mechanism to handle recovery in an object-oriented system, since this mechanism requires the exposition of the internal states of objects.

Vlissides [Vli90] proposed an undo architecture in his **undraw** — a generalized framework for creating object-oriented graphical editors. Logs are used to represent the history of undoable operations. There are two logs: a **past log** that keeps a list of previously-executed commands, and a **future log** that is a list of reverse-executed commands. **undo** will cause

the commands in the **past log** to be reverse-executed and moved to the **future log**, and **redo** will re-execute the commands in the **future log** and move them to the **past log**. This architecture works well with the command based components of **undraw** by supporting arbitrary-level **undo** and **redo** semantics, due to the unique structure of **undraw** and the rich protocol between command objects and other **undraw** objects. Command objects in **undraw** are very interesting objects: they are like messages in the sense that they can be interpreted by other objects, they are like methods in the sense that they are executable, and they are like transactions in the sense that they can be reverse-executed to a previous state. However, the idea behind this structure is similar to the ones used in non-object-oriented systems that record undo-able information at the system level: the object's state change is stored in the executed command object which is then stored in the logs. The abstractions provided by the **undraw** architecture makes this framework powerful in **undraw**-based applications, but it will not work with other object-oriented systems whose components do not implement **undraw** abstractions.

### 2.3.2 Customization

One of most effective modeling techniques is user customization, which is also called demonstrational interface [Mye90], intelligent interface, programming by example [Mye88], or programming in the UI [MW89, Cyp91]. This technique provides facilities that allow the user to change the system behavior to meet the user's needs by creating general abstractions from the specific examples. However, it is relatively difficult to implement customization facilities in UIs. There are no well-known ways for organizing the software, and there are certainly no support abstractions to help implement customization facilities. Most existing systems have been individually crafted.

Common tools to support customization are macro utilities and scripting languages. Macro facilities, such as that of Emacs, record sequences of actions and can replay these sequences. These utilities have limitations because they are difficult to implement in graphical interfaces, and they record low-level actions and have no variables, or conditions that are necessary to create a general abstraction of the user's interactions. Scripting languages are easier to learn than programming languages but they are still a form of programming and pose as great an obstacle to end-users.

Programming by example is an effective technology where the user provides examples and the system infers how the examples should be generalized to create something that is

more general-purpose. Successful systems based on this approach are SmallStar [Hal84], Peridot [Mye88], Metamouse [MWK89], and Eager [Cyp91]. Most of these inference systems use a rule-based mechanism to guess the generalizations. The “condition” of the rule determines if the rule should be applied in the current context, and if so, the “action” fires to assert the generalization. In some systems, rules also contain messages to serve as feedback that the rule is going to fire. These successful customization facilities limit the inferences to a specific domain and each of them has its own application and domain dependent way to record and interpret the user’s interactions.

Peridot [Mye88] is a system for building user interaction techniques, such as menus and scroll bars by using a programming by example metaphor. Peridot is able to infer the user’s action in user interface design by using a rule-based mechanism. For example, after the user places the first few items of a list in a menu, it can infer the placement of the rest of the items in the list. This mechanism is supported by Peridot’s specific architecture.

Metamouse [MWK89] watches user actions and writes a program which generalizes those actions. The whole system is designed specifically to support programming by example. Whenever Metamouse detects a pattern, it predicts subsequent actions and then eagerly reveals its predictions as soon as it can.

Eager [Cyp91] is similar to Metamouse and offers a solution to the task of specifying loops by example. Eager is always on, constantly looking for repetitions in actions performed in HyperCard. It receives information about high-level user events in HyperCard via interprocess communication. Whenever a new event is reported, Eager searches through previous events to find one that is similar. When it detects an iterative pattern, Eager anticipates each next action by proposing a hypothesis. If the user performs the action that matches the anticipation, he/she confirms the pattern, otherwise the hypothesis is rejected.

## **Chapter 3**

# **A User Interface Architectural Model**

The UIMS and UI builder approaches center on the selection and description of the external behavior of the interface. It is common to find published descriptions of UIMSs and builders that extensively describe the supported interaction styles, but devote hardly any space to structural descriptions of interactive systems. Difficulties in implementing modern interface styles via UIMSs and builders have led to the recognition of the significance of structural issues. This thesis is concerned with the internal structure of an interface, rather than the external appearance of the UI. It discusses the software structures with which a given external specification for a UI might be best implemented; it does not address what that specification should be.

This chapter describes a new UI architectural model. It begins with an argument that what is really needed to support UI construction is an application-oriented architectural model that identifies the needs of the UI designer when building application-specific architectures and defines abstractions that address these needs. It then discusses the process and the methodology for designing an interactive system to bring out the kind of support the designer really needs, and presents the UI model with an overview of the abstractions, relating the philosophy behind it, outlining its major elements, and showing how the elements are coordinated to form various UI structures. The chapter concludes with a summary of the proposed model.



### 3.1 The Application-Oriented Model

UI architectures based on the **Seeheim** model are function oriented; they offer pre defined structures that divide the functionalities of an interactive system into three top level components: presentation, dialogue control, and application interface. For example, UMISs usually provide the structure from a functionality perspective with fixed top level components and narrow communication channels between them. Toolkits are component oriented; they offer pre-packaged low-level components like push buttons and dialog boxes that are common in user interaction problems. To achieve a truly powerful interactive application, however, the entire application must be addressed. Therefore, what we really want is an application-oriented model; it addresses what the designer needs to build application specific UI structures. An application-oriented model should define abstractions that help UI designers specify the UI structure by reflecting application demands without being restricted by any pre-defined structure or implementation techniques.

For a dialog-based UI, the difference between the **Seeheim** model and an application oriented model is comparatively small. In both approaches we need to define a parser, and a UI for initiating application actions together with abstract token-stream communication between them. The structure stresses loose coupling among components and specialization of components. However, the difference is significant for a more complex direct manipulation GUI like a drawing editor. A function-oriented model must make many assumptions about the nature of the application data and the way in which the data is manipulated. But an application-oriented model will not need to make such assumptions, because its abstractions should be equally as valid for building a drawing editor as they are for building a dialogue UI, since the same basic components underlie both.

A considerable difference also exists between component-oriented models and application oriented models. The application-oriented approach will give more attention to the flexibility of the system as a whole than to the properties of individual components. Such a holistic perspective suggests that there are system properties related to overall software structure and not just to the properties of specific components, such as help and undo/redo user support features. The basic abstractions provided to the designer should take into account their contributions to the system as a whole.

An application-oriented model does not try to define a high-level structure to problems like a dialogue based UI or a drawing editor UI. Instead, it identifies the components the

UI designer needs when describing UI structures and provides mechanisms that allow the designer to compose these components to meet the application needs.

Although most existing UI development tools (UIMSs, toolkits, and builders) do not include application-oriented structure support, there are a few that do, to various degrees. Three of these systems, listed below, represent current research trends and have had a great impact on this research:

- Nephew [Sze89] demonstrates how to keep UIs separate from the rest of an application, while still providing rich, semantic-based feedback. Its approach is to characterize the kind of information that a UI needs from the semantics, and then define a protocol that allows UIs to access this information. The result is an application structure with three well-defined stages — semantics, high-level dialog, and low-level event handling and rendering — with clear protocols between them. Nephew's successor HUMANOID [SLN92] continues this trend, focusing on automatic generation of UI from designer-provided data models.
- UIDE [FGKK88, dBFM92a] has been applying knowledge bases and tools to aid UI design above the widget level. UIDE shows how design tools and declarative descriptions of application semantics can be combined to help design UIs. For example, UIDE provides a mechanism for making the effect of semantic operations available to the interface, as well as facilities for automating common interface designs and changes.
- Unidraw [Vli90] is a framework for creating object-oriented graphical editors in domains such as technical and artistic drawing, and circuit design. The Unidraw architecture supports the construction of these editors by providing a set of programming abstractions and composition means that the designers need to build new kinds of graphical editors. Unidraw greatly simplifies the development of direct-manipulation-based graphical editors.

These systems have advanced application development technology to a new level beyond the limitations of traditional UIMSs, toolkits, and builders. Nephew and Unidraw have strongly influenced this research, and some of the recent work on UIDE and HUMANOID has similar idea with this thesis. Although these research efforts have paid more attention to applications than just separating out and facilitating UI development, that is, they

no longer treat interface-semantics relationships as unknown and unpredictable, they still treat the application components themselves that way. Furthermore, they treat interface components as well as application components in isolating in the sense that the description for combining the structure of these components is not well supported.

## 3.2 Domain Mapping Approach

UI theory suggests a fundamental separation between the UI, or interactive components, and the computation components in an interactive system. In the UI component, a fundamental distinction is again made between application data and interactive view. Application data (or domain) objects are manipulated by the computational component and presented to users through the interactive view objects. Similarly, interactive views affect application data objects as a response to user inputs. The terms “application data object” and “domain object” are used interchangeably throughout the thesis, and so are the terms “interactive view” and “view”.

Each application has a specific domain of concepts. These concepts and their relationships define the task domain the application addresses. They constitute the domain model that should be represented by application data objects in the interactive system. To present the domain model to users through various media, corresponding media-dependent interactive view objects are defined which specify the interface to application data objects. If the application data objects present a richly-structured domain, then the application’s view objects should reflect that structure in the UI. View objects present the visual structure of the application — an application specifies its appearance by defining and composing view objects. That is, building an interactive application involves the design of both a domain model which reflects a specific domain of concepts and an interface that presents the domain model to the user.

In the design process, the design first defines the application domain model where each data object has an associated set of attributes and behavior. The attributes and behavior of an object are either internal or external. Internal attributes and behavior are meant for use within the application and not exposed in the UI, such as domain data and operations that provide functionality not associated directly with the UI. External attributes and behavior are represented in the UI by interactive view objects of application data that allow the user to control and manipulate domain objects. After the application domain model has been

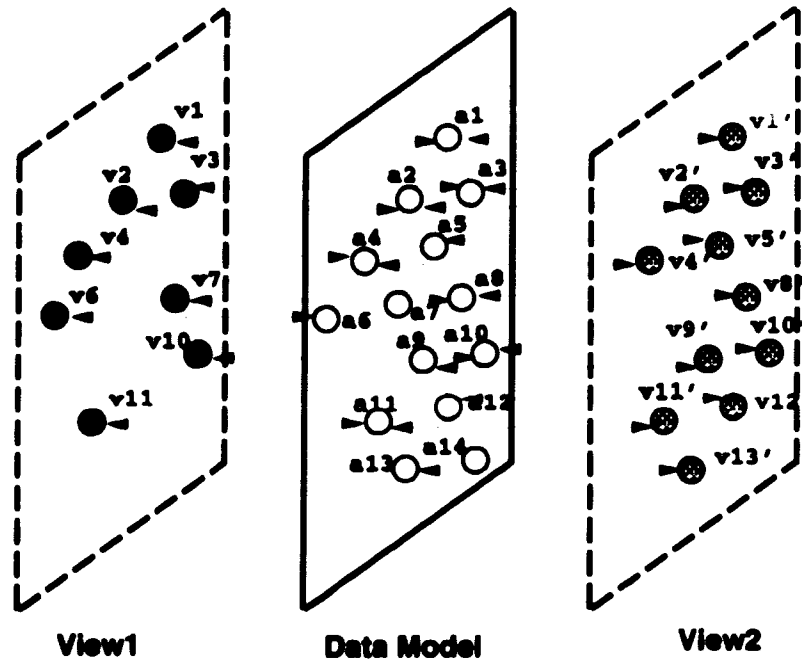


Figure 3: The abstract architecture of an interactive system

completed, the designer then maps the application domain concepts onto the interactive views. The external attributes and behavior of the domain model are represented as the interactive attributes and behaviors of these view objects.

The above domain-mapping design approach suggests that the specification and organization of the view objects should reflect their corresponding application data objects, and interactive view objects are usually as abstract and domain-specific as the application data objects they represent. The UI structure is usually represented as a tree of view objects reflecting the hierarchy of the application data objects. In multiple-view situations, the UI is based on multiple trees. Within a given UI, all the interactive views do not have to be isomorphic to the application data tree. Figure 3 specifies the abstract architecture for an interactive system. Typically, the view trees have more objects in order to supply UI details for control and decoration. Thus many view objects will not be linked to any application data objects (since we are interested in the internal structure of an interactive system, we do not include those view objects (such as menus, buttons, frames, etc) in figure 3).

In summary, in the design process, the designer defines his/her application domain model, and then builds the UI structure by mapping the data objects to the view objects.

These view objects are domain-specific objects that reflect the semantics of applications and match the user's conceptual model. In the implementation stage, these domain specific view objects can be realized by using components of a toolkit. UI toolkits provide basic building blocks for building the UI and it is usually up to the programmer to map the domain-specific objects onto these basic building blocks.

With the above high-level overview, the following are considered necessary operations in designing an interactive system:

- control, manipulate and retrieve domain data and perform other domain tasks,
- reorganize domain data for UI purpose,
- provide multiple views,
- make media decisions,
- provide physical interaction with the user, and
- convert between domain formalisms and interface formalisms.

### **3.3 The Architectural Model**

Part of the design philosophy behind our model is that there is much more to a good UI than just nice-looking widgets, and that higher-level aspects of the design (e.g. task specificity of the interface and the application) are more important than the lower ones. The important goals of the proposed architectural model are to support:

- developing application semantics as well as UIs,
- developing UIs that are well-matched with, and well-connected to, application semantics.

The following describes the new architectural model in enough detail to allow the development of a useful implementation.

#### **3.3.1 Basic Abstractions**

In describing the architectural model the focus was on the common attributes of domain-specific graphical interactive systems. The attributes identified from the previous discussion

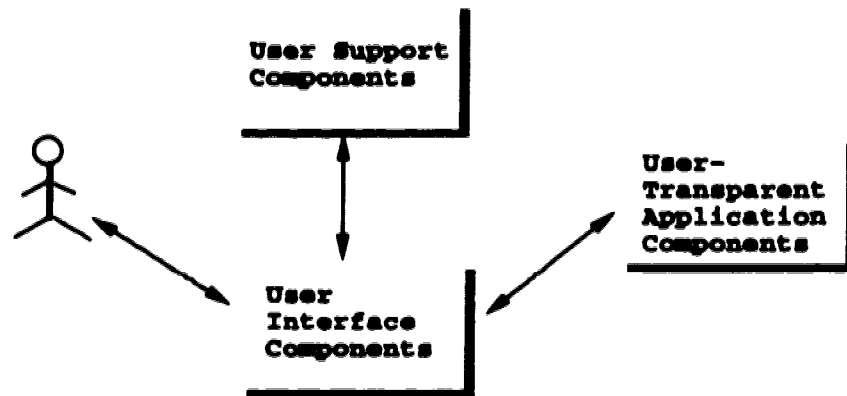


Figure 4: The logical components of an interactive systems

are divided into three parts: UI objects, user-transparent application objects, and user support objects as shown in figure 4:

- **UI components** are directly related to user interactions. They define the input and the output behavior of the system and the related application processing. They implement the concepts that users associate with interaction in the domain. They reorganize domain data for UI purposes, provide multiple view consistency, and choose media for interactions. A UI object includes two kinds of components:
  - **Abstract Application Data (AAD)** components which store, or provide access to, the abstract information (domain data) of the application. They control, manipulate and retrieve domain data and perform other domain-related functions.
  - **Interactive View (IV)** components which present the information to the users, and allow the users to control, manipulate, and retrieve the information. They support the physical interaction with the user.
- **User-transparent application components** are the components of an application that do not interact with the users directly, but contain the functional core of the system, i.e. they implement the concepts of the task domain but as far as the users are concerned, they are not aware of the existence of these components.

- **User support components** are the components that provide services for other objects in the system to implement user support facilities, such as animation, customization, and recovery. These user support features of a UI are system properties, not just the properties of specific objects and therefore are related to overall software structure. They must be addressed at the structure design stage. User support issues will be discussed in the next chapter.

To drive the following discussion, a chess program is used as an example to help explain the concepts defined in the model. Chess was chosen as an example because it is often used in illustrating concepts in the UI literature, and the context is familiar. The chess program from the UI viewpoint is capable of two things:

- checking the legality of user moves according to the rules of chess and reporting special conditions such as illegal moves, pieces removed, etc; and
- responding to a legal user move with an application move.

Using the above abstractions, the chess program has four kinds of objects: **piece**, **board**, **monitor**, and **machine-player**. Each **piece** is an interactive graphical object with position on the **board** as its inner state. A **piece** can send messages to the **board** for notifying when its state has been changed, to the **monitor** for verifying its movement, and to the **machine-player** for informing it of the piece's new position. The **board** is a UI object that arranges **pieces** on the display, forwards the input events to the **piece**, and updates the display when a **piece** is moved. The **monitor** and the **machine-player** objects are user-transparent application objects in the system that do not interact with the user directly, but contain most of the application related knowledge.

The UI objects are the most important part of the system in terms of UI construction, while the user-transparent application objects are the interests of the application designer. The way that these three kinds of components are integrated depends on the abstractions used to structure each kind of component, as these abstractions define the domains which can be handled by the model.

### 3.3.2 User Interface Objects

In an interactive system, it is always true that some of objects are directly related to user interaction, while others are not. The objects directly involved in interaction are called

“UI objects” or interaction objects in the literature. UI objects are software abstractions designed to permit interaction with the user. They represent syntactic or semantic concepts that an interactive system wants to convey to its users, or allows its users to manipulate the system to perform their tasks. Thus, they have behavior: they are observable, they have properties, and their actions are determined by their states and built-in computations.

In general, a UI object includes techniques for either output, input, or both. Output provisions of a UI object define a perceivable behavior in terms of media properties, such as visual or auditory properties. For example, an icon has a specific shape, may be highlighted, and may produce a sound, change color or shape when pressed. Input provisions determine the physical actions the user can perform on the UI object through physical devices such as mouse and data glove. For example, an icon can be moved around with the mouse. From the point of view of the application core that consists of user-transparent application objects, a UI object is an abstraction capable of hiding the details of interaction with the user.

At the UI object level, we are concerned with four aspects of UI objects:

- UI objects as abstractions,
- Architecture of a UI object,
- Composite UI objects, and
- Organizing UI objects into a software structure.

### **User Interface Object Architecture**

In our model, each UI object is divided into two parts: an IV that handles the inputs and feedback of the UI object; and an AAD that implements the application central structure, which defines data structures (attributes) and application semantic processing functions on the data structures. This separation is similar to the MVC. There are several reasons for separating it into two components. First, this approach separates interactive behavior from abstract behavior of an object, so that the orthogonal design of these two components is possible. Second, it supports different views of the same AAD to fit the particular application or to customize interaction style. Lastly, it is independent of UI toolkits. UI toolkits provide various widgets or interaction techniques which define ways to use input



ts” or interaction objects in the literature. UI objects are software abstractions to permit interaction with the user. They represent syntactic or semantic concepts an interactive system wants to convey to its users, or allows its users to manipulate the system to perform their tasks. Thus, they have behavior: they are observable, they have states, and their actions are determined by their states and built-in computations. In general, a UI object includes techniques for either output, input, or both. Output provisions of a UI object define a perceivable behavior in terms of media properties, such as visual and auditory properties. For example, an icon has a specific shape, may be highlighted, and may produce a sound, change color or shape when pressed. Input provisions determine the actions the user can perform on the UI object through physical devices such as a mouse and data glove. For example, an icon can be moved around with the mouse. In summary, a UI object is an abstraction capable of hiding the details of interaction with the application core that consists of user-transparent application logic.

At the UI object level, we are concerned with four aspects of UI objects:

1. UI objects as abstractions,

2. The architecture of a UI object,

3. Composite UI objects, and

4. Organizing UI objects into a software structure.

### **Interface Object Architecture**

In the IOA model, each UI object is divided into two parts: an IV that handles the inputs and outputs of the UI object; and an AAD that implements the application central structure, including data structures (attributes) and application semantic processing functions and control structures. This separation is similar to the MVC. There are several reasons for dividing it into two components. First, this approach separates interactive behavior from the object behavior of an object, so that the orthogonal design of these two components is possible. Second, it supports different views of the same AAD to fit the particular application or to customize interaction style. Lastly, it is independent of UI toolkits. UI toolkits provide various widgets or interaction techniques which define ways to use input

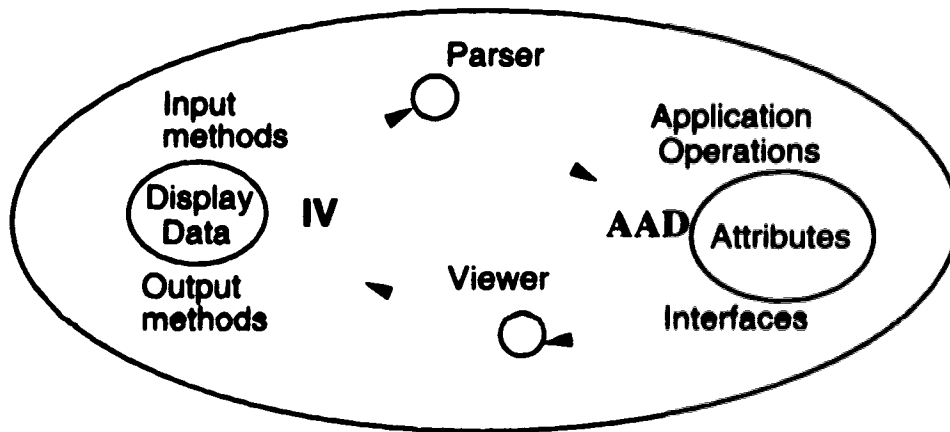


Figure 5: User interface object model

mappings between an IV and its AAD; inputs and semantic operations, display and state. The **parser** maps a sequence of physical inputs to a sequence of state transformations, the **viewer** maps a state change to display changes (if there are several different displays, i.e. one AAD has several different IVs, this mapping will appear in every display). The **parser** and the **viewer** serve explicitly as a translator and a linker. Figure 5 is the logical model of a single UI object. The user's input is accepted by the IV and then forwarded to its AAD as a message through **parser**. The AAD responds to the message by performing the required state transformation and then notifies the **viewer** of the state change. The **viewer** produces a visible display of state by invoking the display methods in the IV.

The standard interaction cycle in a UI object goes as follows: the user takes some action, the **parser** analyzes the input sequence and then notifies the AAD by sending a message; the AAD responds by carrying out the corresponding semantic operations and possibly changing its state and then notifies the **viewer** that the state has been changed in some way; the **viewer**, depending on the nature of the change, updates the display by invoking the display methods in the IV that have access to the AAD's state of interest.

In the chess example, each piece is a UI object that has two parts: an IV component that handles the input and the output of the chess piece on the display, and an AAD component that contains its side and its logical position on the board as its internal state and provides a set of operations on these descriptions as its behavior. The IV component of the piece forwards the user actions to its AAD component through **parser** and the AAD component notifies its change to its IV through **viewer** which then tells the board about its change.

This approach supports the direct manipulation style. In the model, the ordering of physical inputs reflects the ordering of command invocations and there is no intermediate level to reorder invocation in any way, because the inputs are forwarded to AAD directly through **parser**. Display in the model gives a mirror representation of the AAD state. As manipulations are performed on the AAD attributes, the effect of the manipulations are immediately visible in the display. Since each UI object models one individual and small dialogue in the system and because they are independent of each other, a collection of UI objects models multi-thread dialogues.

### **Composite User Interface Objects**

In many cases, multiple UI objects can be combined to form composite objects, which appear to the user as a single object. An example might be a dialogue box, which is a collection of individual widgets that appear to the user as a single interaction entry. In our UI object architecture, a composite can be IV, AAD, or both IV and AAD. In general, the composition is usually referred to as a look-and-feel composition, that is, IV composition in our UI object architecture. If a UI object is declared to be a composite object, it can have children. Composite objects allow the creation of a run-time hierarchy in which the position of a child is specified relative to the position of the parent. If the parent is moved, the child is automatically moved. Therefore, this run-time hierarchy of widgets is based on the underlying windows from which the widgets are constructed.

For composite UI objects, we are concerned about the following facets: management of the geometry of composite UI objects, and use of constraints to express relationships between UI objects. Many of the existing toolkits [IVC89, Fou89, Bor86, Sze88] assume responsibility for managing the interrelationships among some of their widgets which can be used to implement IVs in this UI object architecture and therefore a great deal of new power is introduced by using these toolkits. In this thesis, the concern is more with UI internal structure rather than the geometrical composition of UI objects in the traditional sense.

### **Formal Definition of User Interface Object**

We can give a formal description of the UI object model as follows: a UI object is a tuple  $\langle I, T, S, D, \text{parser}, \text{viewer} \rangle$  where  $I$  is a set of physical inputs,  $T$  is a set of valid state

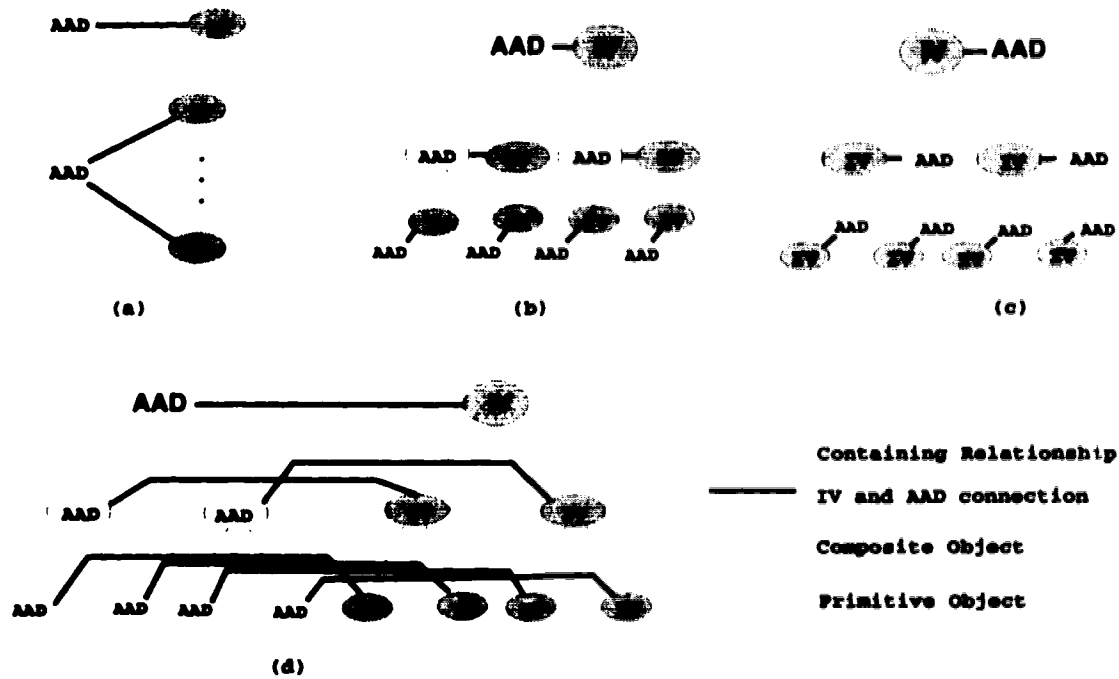


Figure 6: Hierarchical structure of the model

transformations,  $S$  is a set of all possible states the object can have, and  $D$  a set of visible state displays.

**parser** :  $i \in I \rightarrow t \in T$

**parser** maps the inputs into a valid state transformation.

$t \in T : s_i \in S \rightarrow s_j \in S$

$t$  transforms the object state from  $s_i$  into  $s_j$ .

**viewer** :  $s \in S \rightarrow d \in D$

**viewer** produces a visible display of the state  $s$ .

### 3.3.3 User Interface Object Organization – UI Structures

We have identified and discussed the logical components of interactive systems. The issue discussed in this section is how these logical elements are organized into a software architecture.

From section 3.2, the design process of an interactive system begins with building an application domain model which, from the UI perspective, is represented by a collection of AADs, and then the UI is constructed by mapping the application objects onto IVs. Therefore, in a great many cases, the structure of the UI is more complex than a simple pair: an IV and an AAD, since an AAD can have one or more IVs, which provide multiple views of the AAD. On the other hand, an IV or an AAD may be a primitive or a composite, and a composite IV (or AAD) contains other IVs (or AADs) as its components. That is, the UI structure is usually represented as a hierarchy of IVs, which is the domain mapping of a hierarchy of AADs. In a multi-view situation, the UI is based on multiple trees of IVs.

Figure 6 shows the possible logical relationships among IVs and AADs. The well defined protocol of **viewer** and **parser**, between the AAD and the IV maintains the consistency between the AAD and its IVs (figure 6a). When a UI object is composed of sub objects in a hierarchical manner, the hierarchy can be represented in three ways in this model:

- a) in the AADs of the objects, such as in structured graphics where the AADs are graphical objects, this structure is shown in figure 6b;
- b) in the IVs of the objects, such as grouping several IVs which usually coordinate their behaviors, this structure is shown in figure 6c;
- c) in both the AADs and the IVs of the objects, this structure is shown in figure 6d.

In the chess example, chess has the structure of figure 6d. The **board's** AAD is a composite that contains the **pieces'** AADs. The **board's** IV is a composite that contains the **pieces'** IVs.

The above protocols have responsibilities to cover the following three aspects of a UI:

1. for task-level sequencing between the user and the portion of the application domain that depends upon the user,
2. for providing multiple view consistency, and
3. for mapping back and forth between domain-specific formalism and interface-specific formalism.

It is important to mention that the above discussion lists the primitive, logical composition patterns among the UI components that may appear in a UI structure. They are

the primitive patterns that complicated patterns can be built from. They are the logical patterns, and the actual patterns in UIs may not be exactly the same. For instance, a composite AAD has an IV but its component AADs may not have corresponding IVs and their interactions with the user are implemented in their parent AAD. A composite IV may have an AAD but its element IVs may not have AADs.

The composition is done through construction by standard protocols, such as *Insert*, *Append*, *Remove*, *Delete* and child iteration and manipulation operations (*First*, *Next*, etc), between the composition object and its component objects so that the behavior and the attributes of the composite depend on its components and the way they are composed. The compositional mechanism is very important for building up sophisticated and diverse UI objects. We believe that the above composition patterns, though very simple, are powerful enough to describe most UI structures. This approach to support UI structure description is to implement these patterns through well-defined protocols among IV and AAD components which can provide different sorts of composition.

### **Structure Orientation and Function Separability**

The proposed architectural model can support a wide range of structures from data-oriented to interaction-oriented. Data-oriented systems such as database applications have extensive facilities for mapping information flow, to the end users, often with minimal interaction capabilities. These systems typically concentrate on the information being viewed and manipulated by the user. Interaction-oriented systems such as graphics drawing editors have extensive capabilities for mapping user actions into the behavior of the UI, such as controlling appearance, and choosing different interaction techniques for representing the same information. The figure 6b structure is suitable for describing data-oriented systems, while figure 6c is good for most interaction-oriented systems.

The different ways of communicating between the application and the interface define different levels of separability. It is possible to have the whole application functionality defined within the IV hierarchy or AAD hierarchy, which gives a low degree of both functional separability and interactivity between the application and its interface. For example, in a drawing editor application, the functionality of the system would be defined mostly in IV hierarchy by providing sophisticated GUIs, while in a banking system, the functionality would be mostly the manipulation of the application data model and therefore defined inside AAD hierarchy. The structures in figure 6b and figure 6c are suitable frameworks for

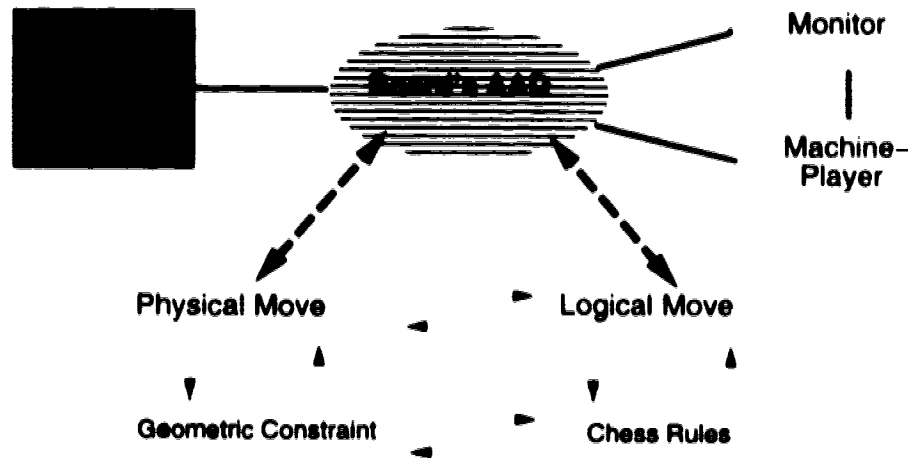


Figure 7: High-level decomposition of the chess program

describing applications with low functional separability and low component interactivity. At the other end of the spectrum, the application functionality can be spread across the boundary of IVs and AADs, giving a high degree of functional separability and component interactivity. Applications like this can be described by structures similar to figure 6c.

### Abstraction Levels

The architecture derived from the above model conveys a hierarchy that can be usefully exploited for defining levels of refinements or relationships in an application. We can describe the abstractions in an application from the high to low levels.

**High level abstraction.** At the top level of the hierarchy, the AAD corresponds to the functional core of the interactive system. For example, in the chess program, the abstraction of the *board's* AAD is shown in Figure 7. At this level of this abstraction, it checks the validity of the user's actions by calling **monitor**, maintains the state of chess playing, and invokes the **machine-player** to respond to state changes. This top-level control has a role similar to that of the dialogue control component in the **Seeheim** model. It bridges the gap between the functional core and the perceivable world in the following two ways:

- It provides mechanisms for indirection and translation between the application processing and the UI. For example, in the chess program, the user's physical

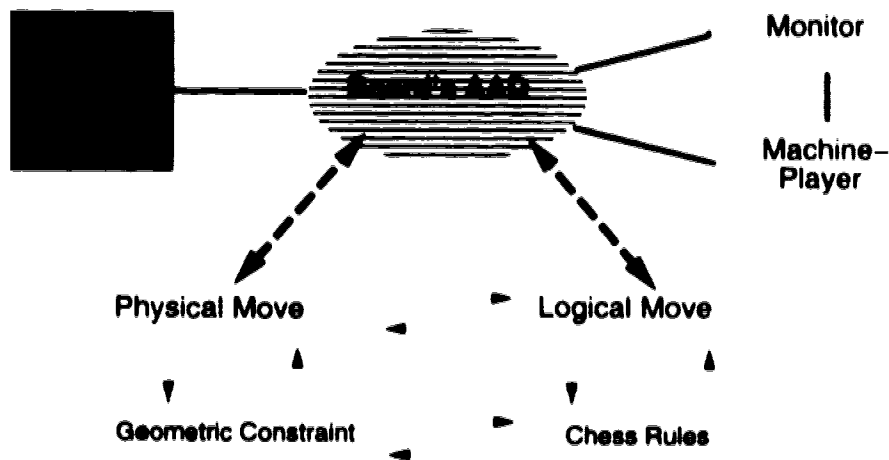


Figure 7: High-level decomposition of the chess program

g applications with low functional separability and low component interactivity. At the other end of the spectrum, the application functionality can be spread across the set of IVs and AADs, giving a high degree of functional separability and component interactivity. Applications like this can be described by structures similar to figure 6c.

### Abstraction Levels

The architecture derived from the above model conveys a hierarchy that can be usefully exploited for defining levels of refinements or relationships in an application. We can describe the actions in an application from the high to low levels.

**Level abstraction.** At the top level of the hierarchy, the AAD corresponds to the functional core of the interactive system. For example, in the chess program, the abstraction of the *board's* AAD is shown in Figure 7. At this level of this abstraction, the user checks the validity of the user's actions by calling **monitor**, maintains the state of the game as playing, and invokes the **machine-player** to respond to state changes. This high-level control has a role similar to that of the dialogue control component in the **hoia** model. It bridges the gap between the functional core and the perceivable world in the following two ways:

- It provides mechanisms for indirection and translation between the application processing and the UI. For example, in the chess program, the user's physical



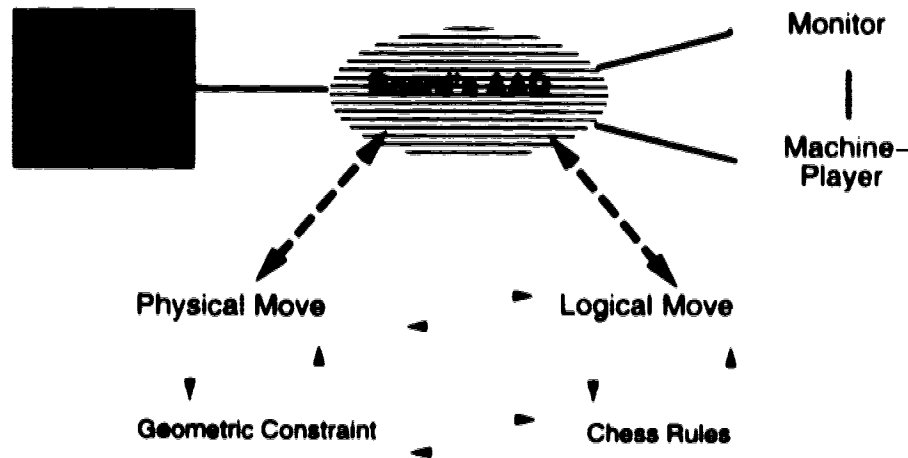


Figure 7: High-level decomposition of the chess program

describing applications with low functional separability and low component interactivity. At the other end of the spectrum, the application functionality can be spread across the boundary of IVs and AADs, giving a high degree of functional separability and component interactivity. Applications like this can be described by structures similar to figure 6c.

### Abstraction Levels

The architecture derived from the above model conveys a hierarchy that can be usefully exploited for defining levels of refinements or relationships in an application. We can describe the abstractions in an application from the high to low levels.

**High level abstraction.** At the top level of the hierarchy, the AAD corresponds to the functional core of the interactive system. For example, in the chess program, the abstraction of the *board's* AAD is shown in Figure 7. At this level of this abstraction, it checks the validity of the user's actions by calling **monitor**, maintains the state of chess playing, and invokes the **machine-player** to respond to state changes. This top-level control has a role similar to that of the dialogue control component in the **Seeheim** model. It bridges the gap between the functional core and the perceivable world in the following two ways:

- It provides mechanisms for indirection and translation between the application processing and the UI. For example, in the chess program, the user's physical

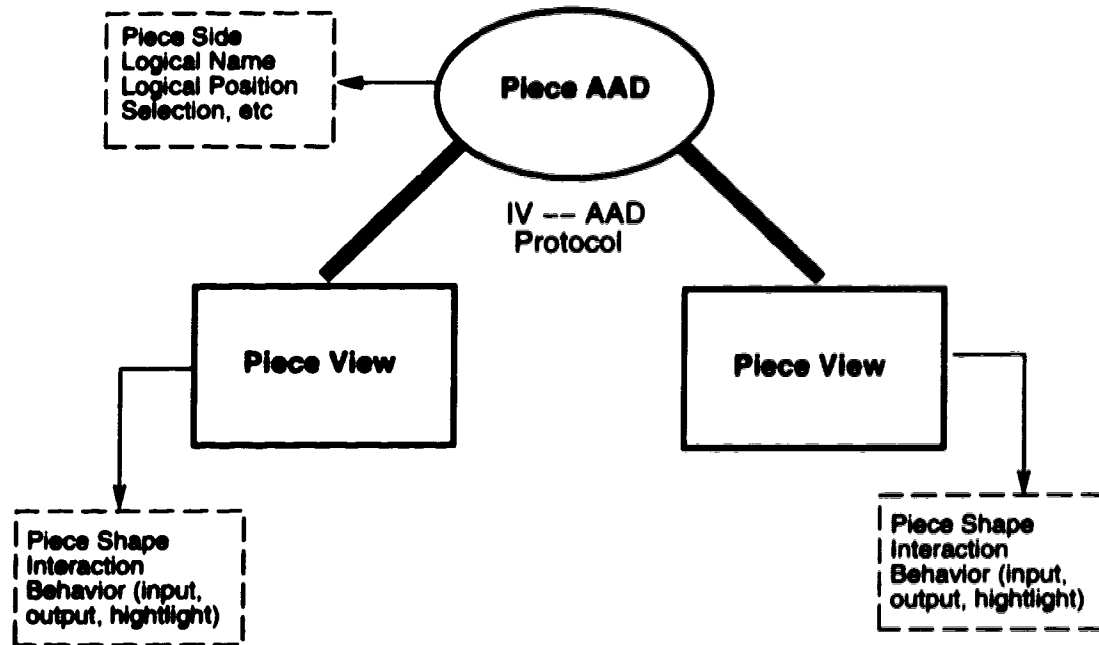


Figure 9: Piece object with two views: component abstractions

and implementation. The combination maps the abstract UI components that define the protocol onto the implementation objects which are provided by available UI toolkits. The high-level components are closely related to the abstraction of the above UI model and are general within the object-oriented paradigm and, on the other hand, independent from the specific details of the implementation. The current implementation is based on InterViews toolkit [IVC89] – a powerful C++ GUI toolkit. We do not present a detailed description of every class; instead, the key classes and their protocols were considered.

### AAD Component

Table 1 lists the AAD protocol's basic operation. One of major responsibilities of an AAD is to handle the communication between an AAD component and its IV component(s). An AAD object defines *Attach* and *Detach* operations to establish or destroy a connection with an IV. These two methods take an IV as an argument. The *Notify* method alerts the attached IVs to the possibility that their state is inconsistent with the AAD's. Upon notification, an IV reconciles any inconsistencies between the AAD's state and its own. The *Update* operation notifies the AAD that some state upon which it depends has changed.

The AAD is responsible for updating its state in response to an *Update* message and then tells attached IVs through *Notify*. The change of an AAD's state is usually caused by invoking one of its member functions which in turn calls the *Update* method. However, there are situations where the synchronization between an AAD's state and its IVs' display is controlled by the AAD's client (IV or other object). The *Update* method is visible to its clients. The *MessageHandle* method is defined as a general communication channel between an AAD and its IVs or other clients which takes a message object as argument and responds to the message by invoking the appropriate member functions. The semantics of this operation is AAD-specific; an AAD typically retrieves information from the message object for internal use. A message defines a request from a client object in a special format just like an event object defines an event. An AAD's client objects include, but are not limited to its IVs. **Tool** and **Command** objects introduced below can also request services from an AAD by passing a message.

Another major responsibility of AAD's protocol is to provide means to organize AAD components to build application data models. An AAD object can be added to (removed from) a composite AAD via the *Add (Remove)* operation. The *GetParent* operation returns the AAD's parent (if any) to allow traversal up the AAD's hierarchy. AAD objects also define a family of operations for iterating through their child AADs (if any) and for re-ordering them. These operations include *First*, *Next*, *Last*, etc which support iterating over the children of an AAD. An AAD may also support maintaining its children in a particular order based on the value of some attribute in its child AADs. The base AAD class only defines the protocol to specify what is desired rather than how to accomplish it.

Finally, in the basic protocol, AADs can communicate with user support facilities to provide context-dependent information. Right now, AADs can set and access its undo facilities which will be discussed in the next chapter via *SetUndo* and *GetUndo*, respectively.

The above protocol is for AAD components in general and can be extended depending on the application semantics. The library has the **GraphicAAD** as an extension to support the structured graphics. Structured graphics refers to a graphics model in which geometric primitives such as polygons can be assembled into hierarchies. Each graphical object has some associated state, such as its geometry and coordinate system specification and application specific-semantics. Primitives can be composed hierarchically to impose structure on the graphics being displayed. Structured graphics can simplify the implementation of direct-manipulation GUIs because it makes creating and manipulating graphical objects easy. The

Return	Operation	Argument	Role
AAD	Attach	IV object	establish connection
	Detach	IV object	destroy connection
	Notify		alert attached IVs
	Update		update state
	MessageHandle	Message	
	GetParent		return AAD's parent
	Add	AAD object	add an AAD as its child
	Remove	AAD object	delete a child AAD
	{child iteration and manipulation methods}		
	GetUndo		
RO	SetUndo	RO object	

Table 1: AAD object protocol

extension of the basic AAD protocols adds graphic-specific operations, such as accessing or setting geometric information, concatenating transformation, etc. The implementation of the **GraphicsAAD** is an extension of the structured graphics provided by **InterViews** so that **GraphicAAD** objects support AAD's protocol as well as the operation of creation and manipulation of graphical objects.

### IV Component

Table 2 presents the basic IV protocol. The base protocol for IV components supports geometry management, rendering, and structuring multiple IVs into an aggregate. The IV protocol duplicates some of the AAD protocol's operations (*Update* and *GetParent*, for a composite IV, *Add*, *Remove*, and those for iteration and child manipulation), adds *SetAAD* and *GetAAD* operations that set and return the IV's AAD, and defines the base functionality of a view, like *Hide* and *Show*. In addition to the communications with undo facility, IVs can also support user customization by having its own customization object (*CusObj*) that will be explained in detail in the next chapter.

The *Draw* method displays the IV by first drawing its appearance on the underlying window and then asking all the contained IVs (if the IV is a composite and has child IVs) to display themselves. The built-in clipping algorithm is set by a composite IV before drawing its children.

A major responsibility of an IV is to handle event dispatching. An IV defines *EventHandle* to interpret the user's input events forwarded by the underlying window system and the semantics of this operation are IV-specific. Events are low-level objects that application code should rarely need to access directly. The *EventHandle* method processes events to the IV by translating events to member function calls or dispatching events to other IVs. The event-driven model, as it has been used for most window-based interactions, is the underlying interaction mechanism. The containing relationships among IVs define the composition of physical IVs, where the relationships among the IVs' *EventHandle* methods specifies the logical composition among the IVs.

IV components are generic objects and therefore, their protocols are independent from the implementation-platform and interaction details supported by the implementation class which is encapsulated by the IV protocol. The base protocol of IVs here only covers the composition of physical and logical IVs. The implementation objects' protocol should specify how to map IV objects onto a screen and receive events from input devices. An implementation object should create a canvas that is bound to a portion of the screen when the IV is mapped and implement many other IV related-operations such as move, raise, and resize. In the current implementation, the underlying window system is X-windows and the IV implementation is build on top of *interactor* class in InterViews. *Interactor* is the base class for all interactive objects provided by InterViews and has a shape variable that defines the desired screen space, an input variable for reading events, and a output variable for performing graphics operations. *Interactor* is a heavy-weight widget that encapsulates almost all the details necessary for performing window interactions.

A selection object is nominally a convenient interface for managing a set of distinguished component views. To support the selection concept as well as scrolling and zooming manipulation, a special subclass of IV called *Viewer* is introduced. A *Viewer* displays an AAD, most often the root of an AAD. It can also process user input events and dispatch them to selected objects. Its protocol supports set and get selection object and scrolling and zooming manipulation on the display.

### Other Components

IV and AAD components are used to express the concepts of the application and define the structure of an interactive system. There is a need for other objects for the application semantics and control aspects of a UI. In our prototype library, we have *user-transparent*,

Return	Operation	Argument	Role
AAD	SetAAD	AAD object	establish connection
	GetAAD		
	Notify		alert attached IVs
	Update		update state
IV	EventHandle	Event object	
	Add	IV object	add an IV as its child
	Remove	IV object	delete an child IV
	GetParent		return IV's parent
RO	{child iteration and manipulation methods}		
	GetUndo	RO object	
	SetUndo		
	GetCusObj		
CusObj	SetCusObj	CusObj object	
	Draw		display itself
	Hide		make the IV invisible
	Show		make the IV visible
	HighLight		highlight the IV
	UnHighLight		unhighlight the IV

Table 2: IV object protocol

*command*, *tool*, *user-support* and *UIShell* classes. *User-transparent* components are the objects in an interactive system that contains the application semantics which is not directly related with user interactions but defines the protocol to talk with UI objects, for example the base class we used to define the *monitor* object in our chess example. The *command* object is the base class for the objects that allow the user to directly access or invoke some of the attributes and methods of a UI object via menu, button, and dialog objects. The *tool* object is a component that supports direct manipulation of a UI object which, together with *command* objects, define the control elements in a UI. The *UIShell* object is the root of IV tree associated with *tool* and *command* objects and dispatches inputs. A *user-support* object implements user support features for a UI object. Detailed descriptions of these objects' functionality and protocols will be presented in the following chapters.

### 3.4 Designing An Interface with the Model

The proposed UI model provides a unified architecture for user interfaces. An interface is composed mainly by IV and AAD components. Given a domain of an application, a designer using the abstractions, namely IV and AAD, first identifies domain concepts that should be presented to the end users, models these concepts with AADs by building an application data model, and then constructs the representation of the application by mapping the AAD hierarchy to an IV hierarchy, and finally, specifies other system components, such as menus, tools, UIShell, etc. Each of the AAD or IV components represents a domain concept and is extended from the AAD or IV abstractions in the framework with application-specific behavior. This approach models the application domain with domain-specific ADDs and IVs that can be refined individually, while the relationships between these components define the application structure. The UISDT, described in chapter 5, supports the above activities as well as prototyping the design.

### 3.5 Summary

The UIMS model supports a high level abstraction used to define the UI. Although these abstractions (or specification techniques) tend to be easy to use, it is usually the case that they are not general enough to allow the specification of all the required facets of the UI, and more importantly they work well only in a limited domain. To get around this problem the designer has to access a lower level programming language or the underlying toolkit which is used to implement the higher level abstractions, and which is general enough to allow all aspects of the UI to be defined. These low level languages (or toolkits) are difficult to use, and it is a long jump for designers from the high level abstractions to the low level languages or toolkits. The toolkits' model, on the other hand, works the opposite way and supports the low level programming abstractions which are general enough to cover most 'look-and-feel' aspects of the UI.

Figure 10 shows the UI design pyramid. The width of the pyramid represents the UI design space supported at that level. The top level is the UIMS environments, although the UI is easy to generate, the designer is limited to a narrow design space. Because of automation in most UIMSs, they provide UI designers with very little control over design decisions. However, generating good UI designs is intrinsically difficult. Principles of good UI design are not yet well specified, (at least not at the implementation level), and cannot

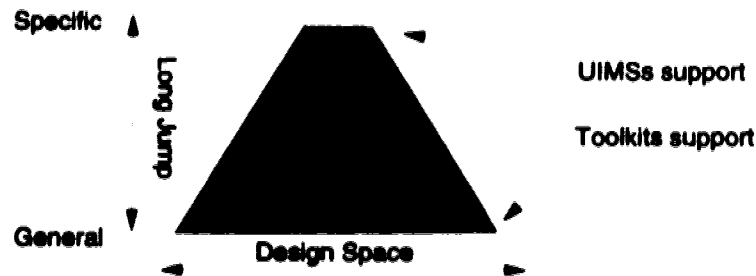


Figure 10: User interface design pyramid

be used to automatically prune the relative large design space. In contrast, at the base of the pyramid, the toolkit or UI builder situation, the specification is more general, and the design space is correspondingly wider. Working at the component level, builders or toolkits force designers to handle too much detail and designers are forced to make design commitments down to the level of individual widgets, and thus distract them from design decisions at the conceptual level. The sharp edge of the pyramid shows the long drop (or jump) between these two levels. This drop (or jump) can often be intimidating for designers and currently there isn't smooth movement between the top level components and low level components.

In this proposed model an attempt has been made to provide a middle level between the two extremes so that the UI parts of an application can blend seamlessly with the other parts, separating them as deemed most appropriate by the designers.

This level is supported in our model by providing a set of middle-level components that top-level components defined in UIMs can be built from, and low-level blocks supported by UI toolkits can be used to realize these components. From the UI point of view, their behaviors are specified by well-defined protocols: the communication protocols between the AAD and the IV and the composition protocols among AADs or among IVs. Any interactive widget can be used as an IV as long as it satisfies the protocols required between the objects, and the same is true for application objects.



## Chapter 4

# User Support Facilities

User support has been considered one of the important issues in constructing good UIs. However, effective support facilities need direct access to the semantics of the interactions. This requires that the support facilities penetrate into the internal structure of the system and the supporting concepts pervade the system. Previous work in this field has been *ad hoc* and highly application specific, so user support facilities, such as undo and customization, have to be constructed from scratch for each application. We have developed two user support facilities: an undo framework [WG91] and a user customization framework [WG93] as part of the UI architectural model. The components in these frameworks can be well integrated into a UI structure and provide support features by cooperating with the UI components discussed in chapter 3. Our goal is to provide frameworks with three key attributes:

- a) They adopt the object-oriented model in which objects encapsulate the common attributes of domain-specific support facilities.
- b) They provide user support abstractions in a broad range of domains and mechanisms that integrate these abstractions into application-specific UI structures seamlessly,
- c) They minimize the effort required to develop domain-specific user support facilities.

### 4.1 An Event-Object Recovery Framework

In section 2.3.1, we briefly reviewed traditional undo approaches and pointed out that they are not suitable for object-oriented systems. In this section, we argue why traditional

recovery approaches are not suitable for object-oriented systems based on the structural differences between non-object-oriented software and object-oriented software. These differences lead to the introduction of a new approach — an event-object user recovery model.

#### 4.1.1 The Problem

From a procedure-oriented perspective, an interactive system can be described as a set of data structures and a set of primitive commands manipulating these data structures. The state of the system is represented by the collection of these data structure values. A command changes the status of some data structures which leads to a change of the state of the system. The system history is described by a sequence of system state changes caused by a sequence of primitive commands. Primitive commands can be considered as transformations of system states. In order to undo a primitive command, therefore, the reverse transformation of that command must be applied to the system state.

Assuming that we have a history list  $H$  that records all the state changes made during the interaction, and a command list  $L$  which is a sequence of primitive commands issued by the user, then,  $H$  describes a locus of the state of the system in the space of all possible system states under the actions of  $L$ . The recovery meta command **undo** can operate on the commands in  $L$ , and reverse the effects of the issued commands to restore both  $H$  and the application to a previous state. We call this approach the **command-oriented** approach, since the commands causing the changes of state are the units of recovery both for storage and actions. This model is based on the assumptions that 1) the individual procedures, which implement the commands, dominate the system structure and 2) the data manipulated by these procedures are globally accessible.

An object-oriented system can be described as a set of objects, where each object is an independent component and provides certain services to other objects. The relationships between objects determine the ways the objects work together to meet the requirements of a particular application. All computations in the system take place as a result of message passing among cooperating objects. Rather than invoking procedures to act on passive data as in procedure-oriented software, messages evoke object activity. The action taken by an object is a function of its interpretation of the message and its internal state. The input events and the messages passed between objects change objects' states and lead to changes to the system state. The effect of an event upon system behavior depends on the event type

and the objects that respond to the event. Therefore, the system state can be represented by the collection of states of all the objects.

There are several reasons why the **command-oriented** approach is no longer suitable as a recovery method for object-oriented systems. First, the **command-oriented** approach requires the system to be able to find commands that cause changes in system state, and find an inverse function for each command that can be used in recovery. Although the procedures that define the interfaces to the objects may be accessible from a system perspective, the implementation of these procedures and the specification of their data structures are no longer visible at the system level. This requirement violates the basic principle of object-oriented design - **encapsulation**, in which the implementation of member functions is private to the object and not available to its users. Second, since the specification of data structures for an object is private to the object, the internal state of the object is not available to other objects, and therefore it is impossible to describe the system state explicitly at the system level. In other words, an object-oriented architecture cannot provide the context-dependent information needed to perform recovery at the system level. The interaction history, the log of information and events that have already taken place earlier, is distributed inside the cooperating objects.

#### 4.1.2 The Event-Object Model

Let us look at how a particular object-oriented system works. In the chess program, suppose the user wants to move piece *A* from position 1 to position 2 in order to capture piece *B* in position 2, the user generates a move event that is sent to piece *A* by downclicking on *A*, moving it to piece *B*, and upclicking piece *A* in position 2. This event causes *A* to move to position 2, then *A* sends a capture message to piece *B*, and *B* responds to this message by moving itself off the chess board. We can see that it is the events and the object's response to these events, either directly or indirectly, that determine the effects of the user's action upon the system. Therefore we can derive an explicit description of the system state changes by recording events that occur during interactions and the actions that the objects perform after receiving these events. That is, the events and the objects responding to the events provide a representation of the user interaction and its effects on the system. This observation is consistent with the fact that most object-oriented interactive systems are based on an event-driven interaction paradigm.

Given the fact that in object-oriented interactive systems the events and the objects responding to the events form the major part of the recovery information, the next question is how to record this information and organize it so that it can be used for later recovery. Our philosophy is that recovery information should be recorded at a relevant level in the system. As we have discussed above, in an object-oriented structure, the system only knows the reason (the event that occurred and the objects that responded to it) for the change, but does not know how the change was made. This makes it difficult to record the complete recovery information and apply recovery operations directly at the system level. For example, in the chess program, to reverse the above move action of piece *A*, the system needs to send the **undo** move message to both *A* and *B*, and let *A* and *B* undo their actions by themselves. In order to record the effects of an interaction, which will be used as recovery information for later undo/redo, it is necessary for the system to know the event and the objects responding to the event which occurred at the system level. Keeping the same spirit, each object involved in the interaction should also know all the other objects (such as its components for a composite object) that have responded to the same event as this object. Therefore recovery facilities should be distributed inside the cooperating objects, since only the objects know what has happened to them. Furthermore, the recovery facility should be able to organize the recovery information in a way that reflects the relationships between the objects so that the control flow in the recovery process implements the correct recovery semantics.

There are several advantages to binding recovery facilities to objects. First, the information concerning recovery, such as an object's state change, the implementation of the member functions that cause the change, and the messages it sends when the object is responding to the event, are only available inside the object. Second, the object knows whether an action is an undo-able action, because it knows the implementation of its member functions. If the action is undo-able, the object knows how to collect the recovery information and later use this information to reverse its effects when the object receives a recovery command. Some actions are not reversible due to semantic restrictions, others may either have no effect on the object state, such as printing or accessing object state, or do not have a general method for recovery. However, the object is the right place to decide what to do, as long as the actions happen inside the object. Finally, with the object's own history log, locating the recovery information is easier and modifying the implementation of the recovery facility for a particular object is simple and localized. In summary, attaching

the recovery facility to the objects supports recoveries that are unique to the domain and context specified by that object.

The recovery facility structure should specify the way that the local recovery facilities work together to record the system interaction history during interaction, and propagate the recovery information in the system during the recovery process. We use the static structure, the containing relationships among objects of the UI, as the structure for organizing local recovery facilities in objects. There are two reasons why this structure is a good way to organize the recovery facilities. First, the system structure reflects the way that a system performs actions to meet its functional requirements, and therefore this structure is a natural way to group the recovery facilities. Second, this static structure is known when the UI structure is provided and hence it is easy for the designer to incorporate the recovery facility into the application structure without changing the existing UI structure.

#### 4.1.3 The Recovery Framework

The recovery framework consists of the following abstractions:

- **Recovery Object (RO)** which is attached to a UI object (either on IV or AAD, depending on which plays more important role in terms of functionality). Each RO maintains the history list for the attached object.
- **Recovery Commands (RCs)** which specify the undo requests such as **undo**, **redo**, etc. an RC is passed to RO as a request and leads to the changes in the history list in the RO.
- **Recovery Information (RI)** is an object which defines an event, an object set, and recovery data. Each item in the history list is a RI.

When a UI object receives an event that causes the object to perform a reversible action and change its state, it will create an RI with the event and the recovery data and then put this RI on its history list. When an RO receives an RC, it responds to the RC according to the current RI: performing the recovery operation based on the event and the recovery data in the RI, and at the same time forwarding the RC to the objects in the RI's object set if there are any. The whole recovery facility in an interactive system contains a set of local ROs attached to UI components. The control mechanism for recovery has two parts:

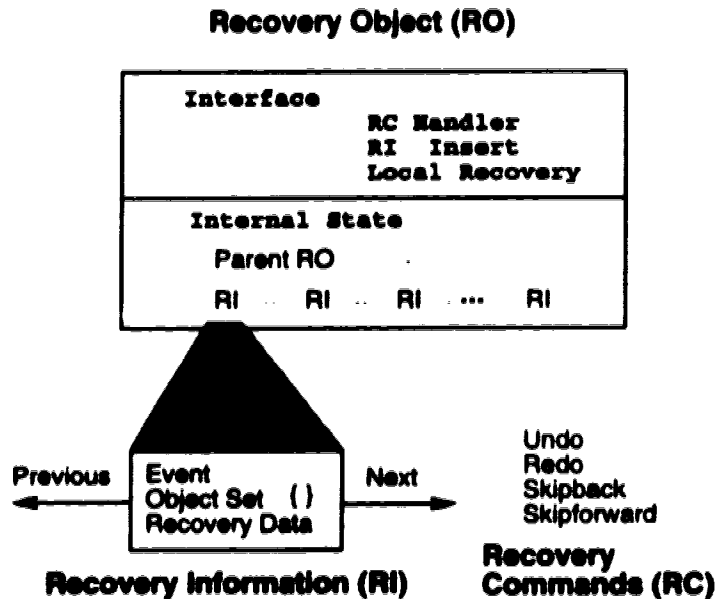


Figure 11: The Components of the undo framework

1. the local control mechanism, which is responsible for making sure that the interaction history is recorded and that the recovery operations are performed in correct order for each UI object, and
2. the global control mechanism, which is responsible for organizing and synchronizing the local mechanisms in the recovery process, and ensuring that the correct user recovery is performed.

### Components of the Recovery Facility

Figure 11 illustrates the components in the framework, and in section 4.1.3, we will describe the object protocols for these components. The recovery framework is composed of a set of ROs that are nested in the UI objects. ROs are organized in an hierarchical structure reflecting the containing relationships among the UI objects in which the ROs are nested. Except for the system (or top-level) RO, which will be discussed later, each RO has a parent RO that is either the system RO or the RO of the composite object that contains the object that the RO attaches to.

An RI object contains three elements: an event, an object set and the local data for recovery. An event is something that happens at a point in time, such as the user pressing the

left mouse button or a message being sent from one object to another. For user interaction recovery purposes, we are interested in the events that change the UI objects' states. The events that are the events of interest depend on the objects that respond to the events. The object set in an RI contains the objects that are components of the attached UI object which responded to the event. In other words, these objects assisted their parent object in the processing of the event. We will discuss how these objects are added to the set later. Since object data structures vary from one class to another, each class should have its own specific recovery data structure. For example, in the chess program, the recovery data for chess piece movement is the distance the piece moved on the board.

The RIs are organized in a double-linked list local to the RO. There is a pointer variable in the RO, called the **history-list-pointer** which points to the current item in the list. When a new RI is added to the list, it will be inserted into the list at the pointer position. The history list can be considered as the internal state of an RO, and the interpreter for the RC's together with an RI insertion method forms the interface to an RO. The RO is built into a UI object as an instance variable.

RCs are the recovery commands that can be accepted by an RO and are applied to the history list to perform different actions. They are recovery commands in the sense that they can be issued by a user, can be applied to an RI, and can change the history list pointer, but they cannot create new RIs. There are four RC's: **undo**, **redo**, **skipback**, and **skipforward**. **undo** reverse-executes the action(s) recorded in an RI and moves the history list pointer back one RI. **redo** re-performs the action(s) undone by **undo** and moves the history list pointer forward one RI. **skipback** and **skipforward** move the pointer backward and forward without any action. The last two operations increase the flexibility of the recovery mechanism. It is important to note that the introduction of the **skipback** and the **skipforward** commands allows a user to have complete control over the previous actions that are undone or redone. It is obvious that this could lead to trouble since the user can undo actions that occurred previously without first restoring the system to the state immediately following that action. Unless the user is clear about what he/she is doing, these two commands should not be issued. In addition if an action has been undone, undoing it again is not allowed.

Return	Operation	Arguments	Role
an RI	UndoInfo	event, object, recovery data	create an RI
Object Set	AddObject	object	add the object to object set
	GetObject		return object set
	SetPrev	RI	set previous RI
an RI	SetNext	RI	set next RI
an RI	GetPrev		return previous RI
an RI	GetNext		return next RI

Table 3: Recovery information protocol

### Component Protocols

The implementation of the proposed recovery framework in an object-oriented paradigm is straightforward. The RI is implemented as a class that has an event, an object set and local recovery data as its attributes. The local recovery data varies from one UI object to another. An application-specific UI object can extend the RI via subclassing to support specific recovery data types. Table 3 lists the RI object protocol.

The RO is also defined as a class and its protocol is described in table 4. *Handle* processes an RC by first looking at the current RI to see if its object set is empty. If the object set is not empty, *Handle* sends the RC to the *Handles* of the objects' ROs in the set. Then *Handle* forwards the RC to its *Undo* for performing the required recovery. *Undo* interprets the RC and determines the action it should take and decides how to do it. This decision is based on the RC, the event type and the recovery data in the RI. Like the recovery data, the *Undo*'s implementation relies on the UI object. An application-specific object should define its own specific *Undo* via subclassing. The *Insert* method will append a given RI to its history list. The *Remove* method will remove the RI from its history list that has the same event identification as the given RI. The *Notify* method is invoked by the children ROs when they create new RIs. They tell their parent RO that a new RI is constructed in response to an event.

The addition of an RO to a UI object is straightforward, since all that a UI developer needs to do is to specify the RO as a member variable of the object, modify the methods



Return an RO	Operation	Argument	Role
	UndoObj	object	create a new RO nested in the object
	Handle	RC'	process RC'
	Insert	RI	add RI to RI list
	Remove	RI	remove RI from RI list
	Notify	event,object	message from children ROs
	SetParent	RO	set parent RO
	Undo	RC'	perform a recovery operation specified by the RC'
	Remove		remove oldest RI from RI list

Table 4: Recovery object protocol

of the object that implement reversible operations so that they create an RI when they are invoked, and define the *Undo* method for the RO.

### The Local Control Mechanism

Each local recovery facility (RO) has its own history list recording the interaction history of the object to which the RO is attached. The RO of the object does not react to events that have nothing to do with the object, nor cause the object to change its state, or lead to an action that is not undo-able. The RO reacts only to events that cause the object to change its state by performing undo-able operations or recovery requests (RC').

Figure 12 diagrams the communication between an RO and the UI object to which the RO is attached. The numeric labels in the diagram correspond to the transmission sequence:

#### 1. Recording the recovery information:

- a) The UI object receives an event and performs an undo-able operation to change its state.
- b) The UI object creates an RI with the event and the recovery data it needs to reverse its action and then sends the RI to the RO.
- c) The RO inserts the RI into its history list and if it has a parent, it forwards a copy of the RI to its parent RO.

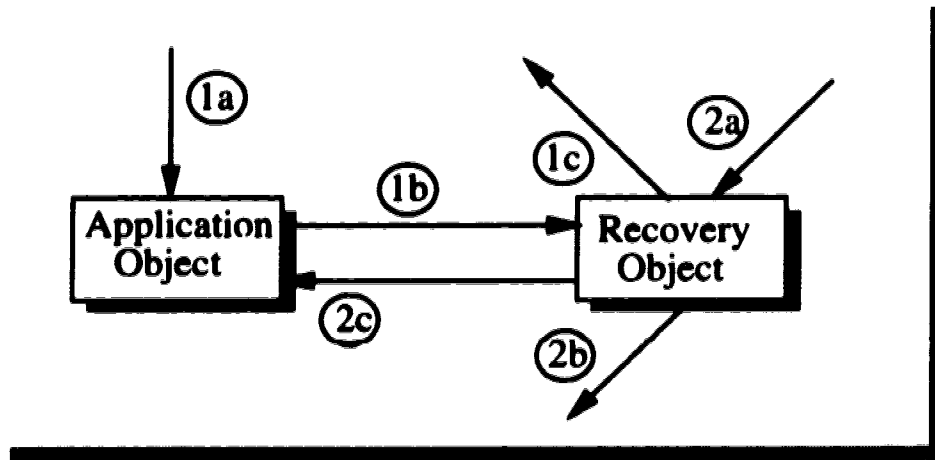


Figure 12: Communication during recording and recovering process

## 2. Performing the recovery:

- a) The RO receives an RC, such as **undo**.
- b) The RO picks up the current RI in its history list and passes the recovery command to the objects that are in the object set of the RI.
- c) The RO performs the recovery operation on the UI object according to the event type and the recovery data in the RI.

To illustrate this process, consider a piece object in the chess program. Suppose the user moves piece *A* from position 1 to position 2. Piece *A* receives a motion event, performs a translation on its position, and creates an RI in its RO with the motion event and the translation distance. When the RO receives an **undo** from its parent later, in this case from the RO attached to board object, it performs the **undo** by applying the reverse translation to the piece's position. At this time the object set in the RI is empty as the piece object is not a composite object.

The objects in the RI's object set are child objects of the object to which the RO is attached. These are the objects that assisted the parent object in the processing of the user's request. There are two situations in which these objects are added to the object set:

- a) When the parent object responds to an event, it sends an event to a child object that causes this child object to perform reversible actions and change its state. As a

result, this child object must be added to the object set of the RI in the parent. When the parent RO receives an RC, it knows that this RC must also be sent to this child object. For example, a composite graphics object contains a set of other graphics objects as its components and when we manipulate the composite, the composite will pass the manipulation event to its components. Later in the process of undoing this manipulation event, the composite must inform its components to undo their operations.

- b) A child itself receives an event from an object other than its parent and as a result it changes its state by performing undo-able operations. In this case, the child is also added to the object set in the parent's RI. This ensures that the parent (and eventually the system RO introduced later) knows all the undo-able operations that have been performed by the user. For example, in the chess program, the board object contains a set of piece objects and when we move a piece, that piece will be added to the RI's object set in the board's RO, so that in later recovery the board knows which piece needs to be undone.

We will give a further description of how these objects are added to the object set of their parent's RI when we discuss the global control mechanism.

In the recovery process, whenever an RC is sent to an RO, the RO first picks the current RI in the list and performs the required recovery operation for the object to which the RO is attached. If there are objects in this RI's object set, the RO sends the RC to these objects. The order in which RCs are sent to the child objects in the object set of the RI is the reverse from the order in which these objects are added to the object set.

### **The Global Control Mechanism**

Conceptually there is a single recovery facility at the system level for recording the interaction history and performing recovery. Though each object has its own recovery facility and the recovery mechanism for the system is composed of these local facilities, from the user recovery perspective, a user's action together with the system's response to this action, is considered to be one recovery unit. That is, a user input event together with the series of events passed among objects caused by this event is interpreted as one recovery event in the interactive system. For instance, in the chess program, the actions of moving piece *A* from position 1 to position 2 and capturing piece *B* at position 2 are interpreted as one action

(moving) for recovery, even though there are several operations among objects. To support this scope of recovery, the event definition for RIs contains a tag field that distinguishes events caused by one user action from the events caused by other user actions.

When an object performs a reversible action(s) in response to an event, it creates an RI with the event and an empty object element set, and then inserts the RI into its RO. The *Insert* operation will first check the current RI (pointed to by its history pointer) in the RI list to see whether its event tag is the same as the event tag in the new RI. If these two event tags are identical (i.e. both of them are caused by one user action), the recovery information in the new RI is incorporated into the current RI, otherwise the new RI is added to the list and becomes the current RI. If the object has a parent, the inserting method notifies its parent RO of the event and the object to which it belongs. The parent RO, after receiving a notifying message from its children, first checks if its current RI has the same event tag. If it does, the RO updates the object set of the current RI by adding the child object to the object set of the RI. Otherwise, a new RI is created with the given event and the child object is added to the object set.

In addition, a special RO — the system RO is introduced to provide a single recovery facility for the user. This RO is nested in the top-level UI object of the system, the UIShell object, that usually contains the control part of the UI (menu bars and tool panels) and coordinates the main interaction loop. The system RO is presented as a pull-down menu containing 4 RCs as menu items (or buttons) to the user and it accepts user requests directly. It records the abstract system history in its history list and performs recovery at the system level. However, it does not perform any recovery operations itself when it receives an RC from the user, because it is not attached to an application-related object. It passes the user's RC to the objects in the object set of the current RI, if the set is not empty.

The actual structure of the recovery facility for a particular interactive system depends on the structure of the application. The local RO can be attached to an IV or AAD depending on the UI structure as discussed in 3.3.3. Figure 13 shows how the recovery facilities work in five typical application structures and demonstrates the information and control flows amongst the local recovery facilities in the process of recording the interaction history and performing recovery. For instance, in Figure 13(1), a user input event goes to composite object 2, which then causes its two component objects 5 and 6 to perform actions. Two new RIs are created in the ROs of objects 5 and 6 when they perform reversible actions and their states are changed due to their actions. Object 2 does not need to create an RI for

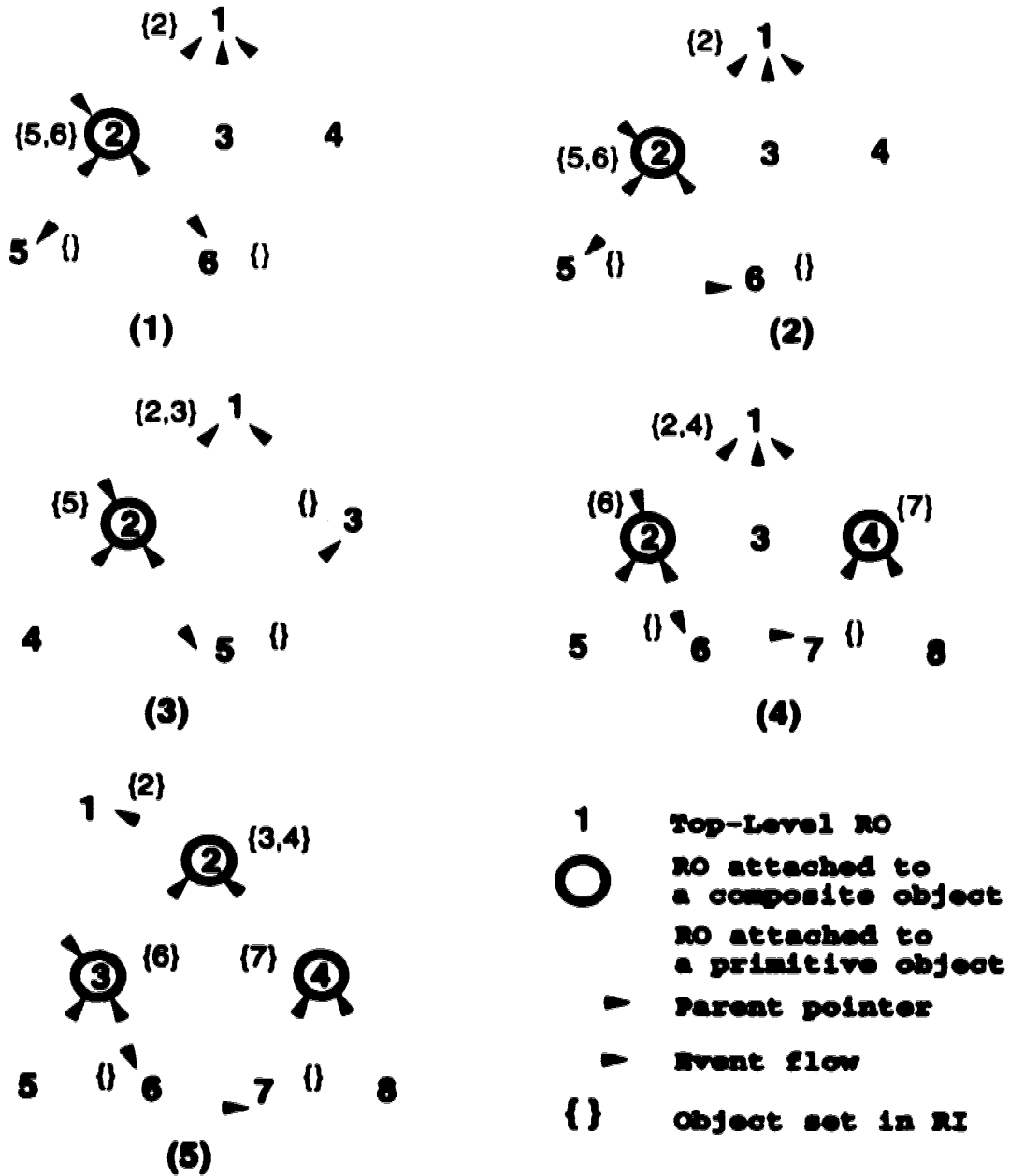


Figure 13: The examples of the recovery facility structure

its RO at first, if it does not take any action to change its own state (without considering its components) after it received the event, or its state is completely determined by its components (it does not have any state except the states of its components). However, when its RO receives messages from its children, objects 5 and 6, it creates a new RI with the user event and an object set containing objects 5 and 6. The order that objects 5 and 6 are added to the object set is determined by the order in which they performed their actions and report to object 2. The reverse order is used when an RC is sent to these two objects. In Figure 13(4), the composite object 2 receives a user generated event and causes its component object 6 to take some action. When the primitive object 6 performs a reversible action on its own state, it sends a message with an event tag to primitive object 7 that leads to a state change in object 7. The primitive object 7 is not a component of object 2, but a component of the composite object 4. Therefore, when object 7 creates a new RI, it causes its parent object 4 to create an RI for the user event. At some future time, when object 4's RO responds to an RC for this RI, it will not perform any recovery action itself, but will send the RC to object 7's RO. The creation of this RI for object 4 is purely for the purpose of control flow in performing the recovery among cooperating objects. The rest of the diagrams can be explained in a similar way.

In the above discussion, we focus on the control structure and mechanism in the recovery facility and describe how the control mechanism can meet various recovery requirements. In an interactive system, however, the actual application semantics may restrict the way the recovery operation is performed in the system. For example, if some objects involved in an interaction can undo their operations and others involved in the same interaction can't, the entire interaction may not be undoable if these objects have dependencies amongst their states. The definition of the RO protocol takes this into account by allowing a child RO to cancel its parent's RI. This cancellation can be propagated along the RO hierarchy to the top-RO, so that the entire interaction can't be undone later.

#### **4.1.4 Performance of the Framework**

The number of recovery objects in an application will be proportional to the number of UI objects, i.e. IVs and/or AADs, in the program, and these recovery objects will be executed each time the user interacts with the system. Each user action will also result in the creation of one or more RI's. Consequently, the storage requirements for ROs and the operations that traverse an RO hierarchy are potentially expensive. Though the actual performance of the

framework in an application depends on the application semantics (storage of recovery data and execution of inverse functions or reassignments), the basic definition of the framework could critically affect the program's performance.

The base class for RO objects requires no memory beyond what the language requires. In the C++ implementation we use, an RO contains several words of storage: a pointer to the table of functions that define the RO's interface, a pointer to its parent RO, a pointer to the history list, a pointer to the object in which it is nested, and two words for itself.

Our scheme for keeping track of the propagation of multiple object changes in the interaction stage and for informing all the relevant objects in the recovery stage is similar to the reasoning process in TMS [Doy79] and constraint-satisfaction processes [DP87]. These processes can be very expensive because they require multiple traversals of the dependency structure. Our approach avoids this problem by using the static structure of the application system. This structure is stable in the interaction process and has a single dependency hierarchy. In our implementation, during the process of state change propagation (recording the interaction history), each RO will only communicate to its parent RO, while in the recovery process, the RO will only send recovery commands to its child ROs, which are recorded in the object set of the RI.

The storage requirements for an RI mainly depend on the recovery data that must be stored in it. The basic RI object, without recovery data, only requires storage for a few pointers. However, when the UI is running, a large number of RIs are generated (at least one RI for each user action). Because the user usually only wants to undo recent operations, only the most recent RIs need to be stored. The *Remove* method of the RO can remove the old RIs from its RI list. The system RO will check the length of its RI list against a predefined maximum length each time a new RI is inserted in the list. The maximum length can be specified by the user or it can take a default value. If it is beyond the maximum length, the oldest RI is removed from the list and its memory is freed. The remove operation also propagates the removal request to objects in the object set of the removed RI, so ROs in these objects will remove the oldest RIs in their RI lists. This process will continue until the object set of the RI being removed is empty. The remove operation is part of the internal implementation of the RO and users are unaware of it.

Compared with a single system history list implementations, our approach does add a certain amount of overhead by attaching ROs to UI objects. However, the major storage and execution time requirements in any recovery facility for interactive systems, especially

k in an application depends on the application semantics (storage of recovery data, definition of inverse functions or reassignments), the basic definition of the framework will not significantly affect the program's performance.

base class for RO objects requires no memory beyond what the language requires. In our implementation we use, an RO contains several words of storage: a pointer to a list of functions that define the RO's interface, a pointer to its parent RO, a pointer to its history list, a pointer to the object in which it is nested, and two words for itself.

A scheme for keeping track of the propagation of multiple object changes in the undo stage and for informing all the relevant objects in the recovery stage is similar to the undoing process in TMS [Doy79] and constraint-satisfaction processes [DP87]. These processes can be very expensive because they require multiple traversals of the dependency graph. Our approach avoids this problem by using the static structure of the application. This structure is stable in the interaction process and has a single dependency graph.

In our implementation, during the process of state change propagation (recording state change history), each RO will only communicate to its parent RO, while in the recovery process, the RO will only send recovery commands to its child ROs, which are in the object set of the RI.

Storage requirements for an RI mainly depend on the recovery data that must be stored in it. The basic RI object, without recovery data, only requires storage for a few pointers. However, when the UI is running, a large number of RIs are generated (at least one for each user action). Because the user usually only wants to undo recent operations, only the most recent RIs need to be stored. The *Remove* method of the RO can remove RIs from its RI list. The system RO will check the length of its RI list against a user-defined maximum length each time a new RI is inserted in the list. The maximum length is specified by the user or it can take a default value. If it is beyond the maximum length, the oldest RI is removed from the list and its memory is freed. The remove operation propagates the removal request to objects in the object set of the removed RI, so ROs of those objects will remove the oldest RIs in their RI lists. This process will continue until the object set of the RI being removed is empty. The remove operation is part of the implementation of the RO and users are unaware of it.

Compared with a single system history list implementations, our approach does add a certain amount of overhead by attaching ROs to UI objects. However, the major storage and execution time requirements in any recovery facility for interactive systems, especially



defines the basic abstractions that the designer can use and extend to build his/hers customization facility in a specific application system. The emphasis is on interaction pattern customization since users usually want to customize patterns of behavior rather than other preferences [Mac91].

The philosophy behind this approach is as follows:

- Users' preferences can be demonstrated by their behaviors.
- Behavior patterns can be generalized from the user's interaction history.
- Behavior patterns can be redisplayed by an active "agent", an object that performs the same task or shows the same behavior.
- The mechanism to record the interaction history, generalize the behavior patterns, and synthesize the agent, can be implemented in a UI.

To have such capabilities in the framework, the following problems must be addressed:

**How to record the interaction history?** To perform the behavior trace, the mechanism must decide what should be recorded and at what level it is recorded;

**What are the patterns?** Given the interaction history, the mechanism should decide how to interpret the behavior trace, i.e., how do we define the class of behaviors the mechanism can recognize;

**How to synthesize the agent?** After identifying the behavior pattern, the mechanism should be able to generate the agent that simulates the user's behavior from the pattern; and

**How to refine the agent?** The mechanism should be able to accept the user's comments or advice about the agent it created and refine the agent.

In the rest of this section, we begin with the discussion of our application-independent approach to building customization infrastructure, then the basic abstractions of the framework are discussed in detail, including their semantics and relationships, and finally prototypes and applications of the framework are presented.

### 4.2.1 Domain Independent Approach

From an application perspective, interactions can be described at three different levels: physical actions with the input device, interaction tasks, the type of information entered by the user [FDFH90], and dialogue. An interaction task is the entry of a unit of information by the user, such as selecting from a menu or picking an object on the display. A dialogue is built on top of the interaction tasks and combines these tasks into a unit task meaningful to the application domain or the problem solving process, such as creating a graphical object on the display or performing a computation on an object. Dialogues are usually application dependent and their definitions or interpretations rely on the particular application domain or the problem solving process.

From a system viewpoint, interactions can also be analyzed at three different abstract levels: input devices, interaction techniques, and UI software. Input devices can be abstracted into the logical devices that shield the system from the details of the physical input devices. Logical devices categorize how user actions are accomplished by the underlying window system and graphics package. Mackinlay *et al* [MCR91] provides a comprehensive taxonomy of input devices. Interaction techniques are ways to use input devices to enter information into the computer and they are built on top of input devices. Interaction techniques are the primitive units of interpretation in UI software and form the basic building blocks of a UI. Widgets in various UI toolkits are examples of interaction techniques that can be used in different applications.

There is a correlation between the conceptual hierarchy of interaction descriptions from the application viewpoint and from the system viewpoint. Specifically, the concepts in the latter hierarchy are implementations of the concepts in the former hierarchy. Input devices make the user actions understandable to the computer system, interaction techniques support interaction tasks (many different interaction techniques may be used for a given interaction task), and the UI software allows the user to carry out different application-dependent dialogues with the system.

A user's behavior is demonstrated by the dialogues he/she carries out with the system, i.e. his/her behavior patterns or preferences are determined by the kinds of interactive dialogues he/she has with the system and the ways he/she interacts with the system. Therefore, the interaction history from the user side is a sequence of interactive dialogues. Most existing customization mechanisms record interaction history at the dialogue level. However, different applications may have different kinds of dialogues depending on the application domain.

The history recording mechanism is application specific if the recording level is at the level of interactive dialogues. This is why existing customization mechanisms are implemented in particular applications. Our approach is to provide application-independent primitives on which the construction of application specific customizations can be built. These primitives together with the common protocols among them provide the fundamental mechanisms and the hooks required to implement various application specific customizations. An interactive dialogue is made up of interaction tasks which are application independent. To have an application independent mechanism, we need to bind customization primitives to interaction tasks. From the system side, this requires that the primitive mechanisms penetrate into the interaction techniques.

In the following sections, we present our framework for constructing customization facilities in GUIs. It has two levels of supporting mechanisms. At the lower level, i.e. at the interaction technique level, there are a set of application-independent customization primitives that are bound to individual interaction techniques so they can store customization information and play back user interactions for each interaction technique. At the higher level, there are a set of abstractions that coordinate the lower level primitives to perform three major functions:

- behavior trace that records interaction history by gathering customization primitives,
- modeling that generalizes user behavior patterns and rearranges the customization primitives to form a simulation agent, and
- simulation that performs the tasks on the behalf of user.

These high-level abstractions are application related and they will be extended to include application-specific semantics and requirements when the framework is integrated into an application.

### 4.2.2 The Architecture

In designing the architecture of the framework we focused on the common attributes of user customization. The attributes we have identified are reflected in four classes of objects:

**Customization object** is a primitive bound to an individual interaction technique that provides technique-specific recording and replay mechanisms for user interactions. For

example, a menu-specific primitive can record selections of a menu item and replay this selection.

**Logger** watches how the user accomplishes a particular task or performs routine tasks through communications with the customization objects bound to interaction techniques involved in the user interactions. It records the user interaction history by grouping customization objects and the corresponding interaction techniques. It is used to represent the user behavior and interaction history.

**Modeling component** interprets the recorded user behavior and identifies his/her behavior pattern or intentions according to pre-defined “models” or templates. A model is a generic behavior pattern that describes families of similar patterns in user interactions. We define a specific user interaction behavior as an instantiation of a model.

**Simulation agent** is created based on the behavior model and acts as an active agent in the UI. It has a control structure defined by the model and the semantics provided by the logger. It can perform tasks on behalf of the user.

The idea of models is a useful design aid for exploring the architecture of customization before having to contend with details of specific behavior patterns. There are a few application independent behavior patterns such as repeating a sequence of actions or performing the same operation several times. These patterns can be defined as models, the structures used to describe the behavior patterns, such as sequential control for routine tasks and loop for iteration tasks. There are also application dependent behavior patterns, where their interpretation needs knowledge about the task domain. The modeling component contains an extensible set of these models and interprets the user interaction history in order to transform the models and fill in the model with the knowledge of a user's behaviors. This modeling approach is similar to application-dependent planning [Geo87, Rob83]. The models correspond to generalized plans, the interpretation of a user interaction history is the process of plan recognition, and agent creation and execution is the process of plan synthesis and execution. We will discuss the details of these components later. Since the task domain and therefore the behavior patterns usually are well-defined ahead of time in UI design, we can also define them as application-dependent models with rules and conditions specific to the application. These models define the area that the customization facility targets in assisting the user in customization. They are the skeletons of the desired behavior patterns which can be fleshed out with the interaction history of the user.

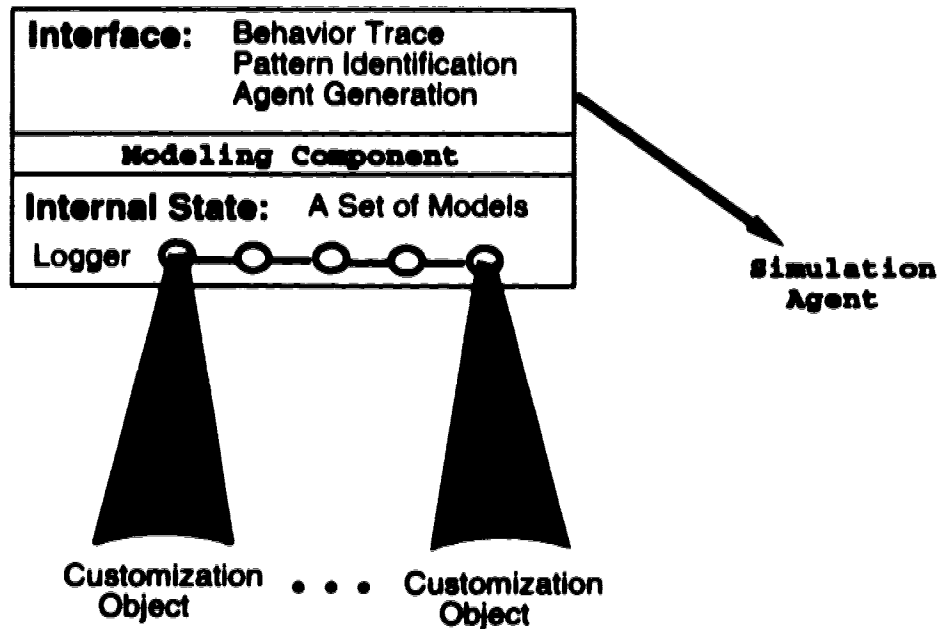


Figure 14: Structure of the customization framework

Figure 14 shows the general structure of the user customization facility.

### 4.2.3 Basic Abstractions

By identifying the common attributes of direct manipulation GUIs, we identified the following abstract interaction components:

**IV components** represent the elements of the task domain. A UI's main objective is to allow the user to manipulate them to perform tasks. For example, in a chess program, a piece's IV on the board is an interactive component that the player can move on the board.

**Command** components are special interactive components that present some control aspects of UI objects to the end user and define operations on UI objects. When a Command object, such as menu item or a button, is selected, it applies the specified operation to the selected component.

**Tool** components are similar to command components that present manipulation aspects of UI objects to the user and support direct manipulation of the interactive components.

A tool defines a particular way to manipulate components and its selection makes this manipulation mode active. For example, selection of a **move** tool allows the user to move objects in a drawing area, and selection of a component tool allows the user to instantiate an instance of that component in the drawing area.

### **Customization Object and Logger**

By definition, customization is the act of altering something to meet the unique requirements of a particular person. Therefore customization can only be achieved if we are able to reason about user behavior. To perform such reasoning, we must have a mechanism whereby we can gain knowledge of the user. A customization object gathers data about the user interaction with a particular interaction technique. Acquiring user knowledge through customization objects is therefore application independent and transparent to the user.

The purpose of customization objects is to record the user actions on the individual techniques and replay the user actions on the techniques later. Customization objects are special objects that are created by individual techniques, stored in the logger as part of the behavior history, interpreted by a modeling agent, and executed by a simulation agent. For the three abstract interactive components mentioned above, we define three corresponding customization objects (CusObj): a CusObj for IV components that records the effects of the selection and the results of manipulations, and replays them back on the component; a CusObj for command component that records the selection and plays the selection back; and a CusObj for tools that records if this tool has been selected and how it is used after it is selected.

The Eager system [Cyp91] for recording high-level events is similar to our approach. Eager also works at the abstract level rather than with primitive user actions. Our approach is more general in that interaction techniques and interaction tasks are well-recognized concepts compared to the high-level events in Eager. It is also more flexible in that for a complex interactive component composed of primitive techniques, we can simply assemble primitive customization objects to form a composite customization object.

The logger represents the interaction history in terms of customization objects. They are organized in a linear list, when an interactive component is involved in interaction, a customization object for the component is created as an item in this list. The logger can then be used by the modeling agent to interpret and transform the simulation agent.

### **Modeling and Simulation**

The modeling process is perhaps the most important stage in customization. The modeling component contains a set of pre-defined models as its knowledge base for interpreting the given logger. The model encapsulates the mechanism of a particular behavior pattern through its protocol. The modeling component maps the interaction behavior in the logger to one of models and transforms the selected model into an executable object, a simulation agent in the UI. The transformation starts by analyzing the user behavior and then incorporating the customization objects in the history list into the matched model, and creating a simulation agent that the user can use as if it was part of the UI. The model can define iteration and conditional structures with variables as in Peridot and Metamouse. These structures are matched by analyzing the recorded behavior history with variables replaced by corresponding interactive components.

Figure 15 shows the communication between different components in the customization process. This process has three phases:

1. The logger watches the user as he performs a series of operations on the system and saves the trace of user-system interactions.
  - a) the modeling component receives a user signal indicating the start of his example;
  - b) the modeling component puts interactive components in trace mode;
  - c) when an interactive component is engaged in user interactions, it creates a related customization object;
  - d) the customization object records user interactions on the interactive component;
  - e) when the interaction is done, the customization object is sent to the modeling component.
2. The modeling component maps the trace to one of the pre-defined behavior models and the trace is transformed into an executable agent.
  - a) the modeling component receives a user signal indicating the end of the trace and starts interpreting the recorded behavior;
  - b) the modeling component replays the recorded behavior to accept user advice on refining it;
  - c) the modeling component creates generalized behavior as an active agent.

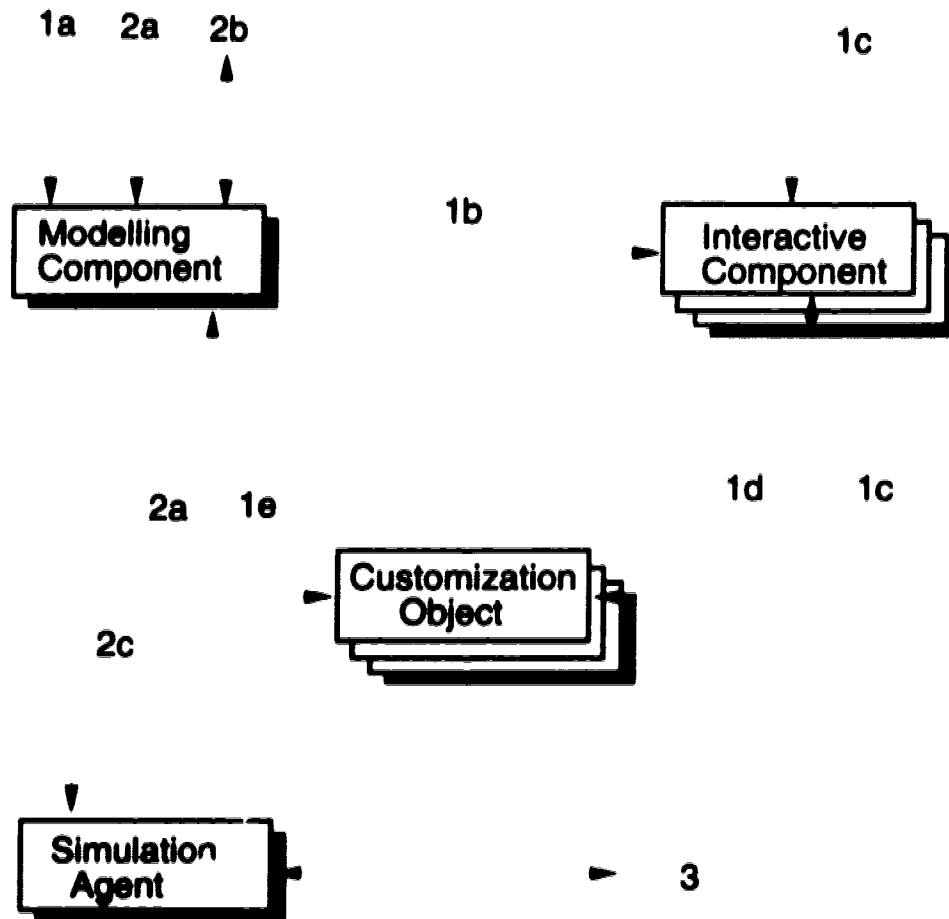


Figure 15: Communication between objects during customization

3. The created simulation agent is integrated with the rest of UI and acts like a new menu item.

We use lazy analysis of the user behavior pattern. Eager uses an eager evaluation strategy where pattern identification analysis is performed on each event as it is logged. We can easily implement eager evaluation in our framework by invoking the interpretation process each time a customization object is recorded in the logger. We think lazy evaluation is more efficient since user patterns may change several times during a trace.



Operation	Arguments	Role	Return
CusObj	interactor	create a CusObj	a CusObj
Store	event	accept event	
Filter	event	filter event	boolean
Playback		replay	

Table 5: Customization object protocol

#### 4.2.4 Prototype

We developed a prototype to test the viability of the architecture. Subsequently, the prototype was used to build two customization facilities in two different GUIs for experimental evaluation of the framework. We focus on the salient aspects of the prototype implementation. We do not present a detailed description of every object; instead, we consider the key objects and their protocols.

#### Customization Object

The customization object's protocol defines basic operations for recording, maintaining and replaying user actions on the interactive components: IV (IVCus), command (CommandCus), and tool (ToolCus). It defines and manages information necessary to replay user actions on the component. Table 5 lists basic operations in the customization object's (*CusObj*) protocol. A CusObj is created by passing an object for which it provides customization service. It communicates with the object to acquire the event information through a *Store* protocol. A CusObj maintains an event list that records the events necessary to repeat a user's actions on the interactive component. The *Playback* protocol lets the interactive component replay its behavior by supplying the recorded event list. Since not every event passed to an interactive component during interaction is important for later replay, the *Filter* operation implements a component dependent event filter to weed out trivial or irrelevant events.

CommandCus, ToolCus, and IVCus are defined as subclasses of CusObj that provide commands, tools, and IV components with specific customizations separately. With the dynamic binding mechanisms in C++, these objects are derived from CusObj with particular behavior defined by their own *Filter* and *Playback* methods.

Operation	Arguments	Role	Return
Trace	CusObj	set trace mode	
AddItem		append CusObj to logger	
Simulate		replay the logger	
Interpret	Model	identify pattern	a Model
Create		create an active agent	a Agent

Table 6: Modeling component protocol

### Modeling Component

Conceptually, the modeling component is the only component that is known to the user. It encapsulates the three major functionalities for customization, namely trace, modeling, and simulation. It contains an extensible set of models that are the pre-defined behavior patterns the customization facility can support. It also stores a logger as its internal state. It interacts with the user through menus that allow the user to customize the UI. The modeling component protocol is shown in Table 6. The *Trace* operation allows the Modeling component to watch the user during interaction. It informs all the techniques involved in the interaction to record their local customization information by creating technique-specific customization objects. These individual techniques report their behavior to the modeling component using the *AddItem* protocol that appends the given CusObj to the logger.

The *Simulate* operation accesses the logger, invokes the *Playback* method of each CusObj, in the order they were added to the logger, to repeat the interaction process that has been recorded. The purpose of this operation is to give the user a chance to confirm his behavior pattern, and allows the customization facility to accept the user's comments in order to refine the generalized pattern. The user can interactively verify the replay and the simulation process and can modify the logger. The previous undo facility is used to reverse the effect of undesired actions during replay.

The *Interpret* operation implements behavior pattern identification and model generation. The modeling component maps the user's behavior in the logger to one of the pre-defined models by analyzing the logger and comparing it with the models. Different models need different matching algorithms. The default implementation of the *Interpret* operation is to match sequential control for routine tasks and loop for iteration tasks.

The sequential pattern is the simplest model that is used to describe the control for routine tasks. The model defines an array of CusObjs whose number and contents depend

on the interpretation, and a for loop control structure which invokes each of the array elements sequentially. The interpretation process moves the CusObjs out of the logger and instantiates the agent from the model by filling the array of elements with these CusObjs.

For iterative patterns, the model describes a loop structure for certain tasks, the matching algorithm analyzes the CusObjs in the logger to find similar CusObjs where the interactive components involved are the same type, and the contents involved in the interaction fit into some regular pattern (e.g. open two ".c" files). The definition of similarity can be varied depending on the applications. For example, it could be the same type of events, the same response from the interactive components, or the repetition of the same action sequence.

The *Create* operation creates an agent from the matched model and integrates it into the UI so the user can directly access it through a menu. The agent is an object that looks and behaves like a menu item and can execute a user behavior pattern by invoking CusObjs' *Playback* methods.

#### 4.2.5 Experience

We installed our framework in two different graphical applications to evaluate both the concepts and the prototype implementation. Our aim was to demonstrate that the framework supports diverse domains, and reduces development effort. The graphical applications are a drawing editor and a chinese chess program. We chose these two applications because each represents a typical application and has different domains and customization requirements. We discussed the drawing editor's application in this section and the chinese program application in chapter 6.

The drawing editor, called *idraw*, is built on top of InterViews and is distributed with the InterViews library. It is similar to MacDraw in that it provides an object-oriented, direct-manipulation editing environment for producing drawings and diagrams. By using *idraw* to produce drawings, individual users develop their own drawing patterns for their drawing tasks. For example, the background of figure 12 is a particular pattern that was created by first instantiating a rectangle with no border, filling it with a fill pattern, then instantiating another rectangle on top to it with dash-line border and non-fill pattern, and finally grouping them together. To create such a background, the user needs to select menu items and tools seven times. There are many other similar patterns that have been developed by *idraw* users through using it, such as laying graphical objects on the screen

with certain alignment, etc. We have installed our customization framework in idraw. The customization patterns we need are the default ones – sequential control for routine actions and loop control for iterative actions. The CommandCus and ToolCus can be used directly by idraw menus and tools. The IVCus is extended to record the user operations on viewer (a subclass of interactor) between his/her selections of tools or menus. This extended idraw has been used to produce various drawings with customized interfaces.

#### 4.2.6 Summary

We proposed a framework for building user customization facilities in GUIs implemented with UI toolkits. The framework simplifies the construction of domain-specific customizations by providing programming abstractions that are common across domains. It defines three basic abstractions:

- primitive customization objects encapsulate the recording and replay mechanisms for individual interactive objects in a domain, i.e. they are embedded in IVs;
- the modeling component records user's interaction, identifies the user's behavior pattern and creates a simulation agent;
- the simulation agents perform the tasks on behalf of the user.

This framework has been integrated into our UI model, so as Undo framework.

Note that this framework does not present a solution for particular types of customization or any application system. It also restricts its application on direct-manipulation based GUIs and encapsulates a few simple application-independent customization mechanisms. The framework is based on a task-oriented approach instead of the solution-oriented approach that have been used by many existing systems. Its basic abstractions support low-level customization tasks that most customization facilities need rather than high-level application-dependent customization mechanisms that are provided by other customization systems. This framework is not a complete customization facilities, rather it provides an infrastructure or a skeleton that UI designers can use and extend to build their application-specific customization facilities.

The hard technical problem in customization has always been how to infer the right generalization. An effective customization requires generalization to turn a trace of user interaction into a useful agent. Although building a successful system requires extensive

knowledge about the application domain (for example Metamouse succeeds in its well defined limited domain) and our framework does not address this issue directly, we try to define an organization or infrastructure to minimize the difficulty in solving the problem.

## **Chapter 5**

# **UISDT — A User Interface Design Tool**

The **User Interface Structure Design Tool (UISDT)** is an experimental design tool that allows the designer to produce a UI using a graphical editing metaphor. UISDT, a tool based on our model, imposes many more constraints on the designer than the model itself, since it serves different functions, it embodies a model of an interactive system (the UI architecture model) as well as a model of a design process. The goals and strategies of UISDT's design are:

- Embed our application-oriented model in the design tool. This UI model is the infrastructure upon which an application-specific structure is constructed. The middle-level components provide basis for application-specific objects. The default values in middle-level components as well as the protocols provide a certain level of design automation.
- Separate the UI structure from the design of the external appearance. UISDT emphasizes UI structure design instead of look-and-feel design as in most UI builders.
- Adopt an object-oriented model for UI design. UISDT supports designers in defining object models of their applications by providing abstractions and implementing applications by producing prototypes.

- Provide adequate balance between control over design flexibility and design automation. UISDT strikes this balance between given designers control over the details of a design, and providing a high level of design automation.
- Provide an exploratory environment for UI design. The UI architectural model is a reference framework that guides the process of UI software organization.

## 5.1 UISDT in Action

UISDT is first described in relation to a concrete example. This section illustrates the use of UISDT to create an interface for the chess program example that used throughout this thesis. Space limitations require that some details be left out, but a comprehensive explanation of design process for chinese chess appears in chapter 6.

UISDT screen is shown in Figure 16. There are six parts: a title bar on the top, a status line immediately below the title bar, a menu bar immediately under the status line, a toolbar immediately below the menu bar, a toolbox at left of the screen, and a drawing area. The title bar gives the name of UISDT and is common to most X-window based applications. The status line displays the file name of the application design and the connection mode. The drawing area takes up much of the center of the screen.

Selecting items from the pull-down menus listed on the menu bar is one of most common ways to unleash the power of window-based applications. The same is true for UISDT. The menu bar gives a designer the tools needed to develop, test, and save his/her application. The File menu contains the commands for working with files that go into the application. The Edit menu contains many of editing tools that help the designer in editing his/her application, such as commands for generating the code of the ongoing design, testing the application, and undoing the editing operations. The View menu allows the designer to have multiple views of several versions of his/her ongoing application design. The Custom menu lets designers to customize their design patterns or create macros for their routine design steps.

As is becoming more common in window applications, UISDT uses a toolbar to let designers activate common tasks without using the menus. Since every item on the toolbar also has a keyboard equivalent the designer can choose his style of interaction. We will explain the items on the toolbar later. The tool palette (toolbox) contains the 14 basic

component tools for a designer to develop his/her application. The details of the toolbox are discussed later in this chapter.

The first step in developing an interactive system within UISDT is to specify the application model — in other words, to design the system structure. What kinds of domain objects do you want to model? What are the relationships among them? How are these objects delegated to a user? What kinds of controls does a user have over these components? With UISDT, application models are created by dragging objects out of abstractions supplied by UISDT (the left-hand toolbox in Figure 16) and pasting them together. In Figure 16, the designer has created a **Board** AAD — a composite graphic AAD that models the board concept and various piece AADs that model piece concepts. The designer drags the desired component in the toolbox (in this case the composite graphic AAD) and places it in the drawing area.

The designer defines the application-specific component's behavior by using the refine tool from the toolbar and performs in-place editing on the component. By dragging a component tool from the toolbox and dropping it on the drawing area, the designer tells UISDT that he wants to create a component derived from the tool or that his domain component is based on the abstraction that the tool represents. By engaging the refine tool, the designer can view as well as edit the properties and behaviors of a component. The designer presses a mouse button on top of the component and a new window opens up with three categories of information, properties, behaviors, and relations. The designer can pursue each category by opening the sub-windows. The details of this process are presented in the following sections. In general, the designer can change the default behaviors and add new behaviors through the component's in-place editing feature.

The **Board** AAD contains various pieces, and the **Piece** AAD is the base class for various pieces, each of which differ in appearance (bitmap) and behavior (moving and capturing rules). A containing relationship is established by connecting a composite AAD (board AAD) to a primitive AAD (a piece AAD), while a subclassing relationship is created connecting two classes, such as **piece** and **pawn**, with the inherit relation tool selected from the toolbar. By defining AADs and the relationships among them, we specify the hierarchy of the AADs in the chess program, its application data model.

Next, the designer creates two IV components: **Board IV** and **Piece IV**. The **Board IV**, a composite IV, defines the interaction behavior for the **Board** AAD that displays the board on the screen as well as pieces (by invoking the **Piece IV**'s draw method), and receives the



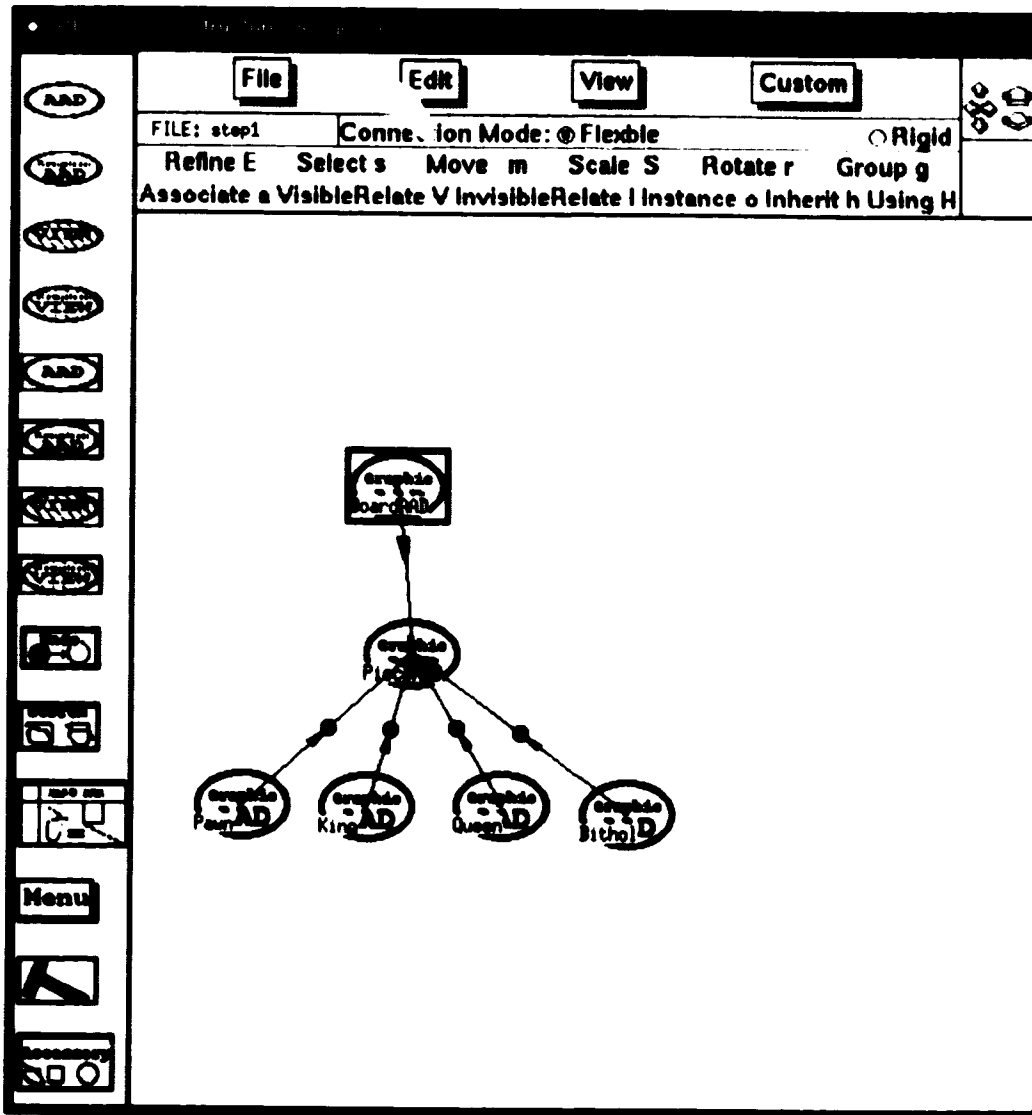


Figure 16: Building application model

user's input. The **Piece** IV specifies interaction behavior for the **Piece** AAD. The process of creating components and establishing relationships is similar to drawing a picture with a drawing tool. The most important relationship is the relation between AADs and IVs. The result is shown in Figure 17.

Finally, the designer creates a top IV component — a subclass of **UIShell** that provides a top-level view for the chess program by containing the **Board** IV and a set of menu buttons (*newgame*, *quit*, etc). The designer uses the same approach to define the properties, behaviors, and relations of these components as in specifying AAD and IV components. The result is shown in Figure 18.

This section mainly described building a UI structure in UISDT, leaving out the details of refining each component (its behavior as well as semantics). A more detailed description of how to use UISDT to build applications appears in the next chapter. In the rest of this chapter, we describe UISDT by first describing the design process it supports, then the components it provides, and finally its prototyping capabilities.

## 5.2 Architecture of UISDT

UISDT is a tool that supports the designer in developing a UI based on an object-oriented model. It improves on other UI design tools that generate parts of a UI from data models, such as tools producing widgets from data types [dBFM92a, Mac86], or syntactic components and external layout of a UI from dialog and UI component specifications [Mye90, SG91]. In particular, it relies on the object model of the application and takes advantage of the relationships among the objects.

User interface structure construction is the primary interest. Unlike most UI builders, UISDT is designed to support the internal structure of the UI, not the external appearance or layout of a user interface. UISDT supplies the components of the proposed UI architectural model for the designer to specify his/her domain concepts and provides composition mechanisms for building his/her application-specific UI structures.

UISDT divides the development task into two major steps: first, building object models in which designers are engaged in specifying application object models by using and extending UISDT's components, and second, prototype generation in which UISDT creates the implementation from the object model while designers are hidden from the code generation process. Object models contain more information than previously used for UI generation.



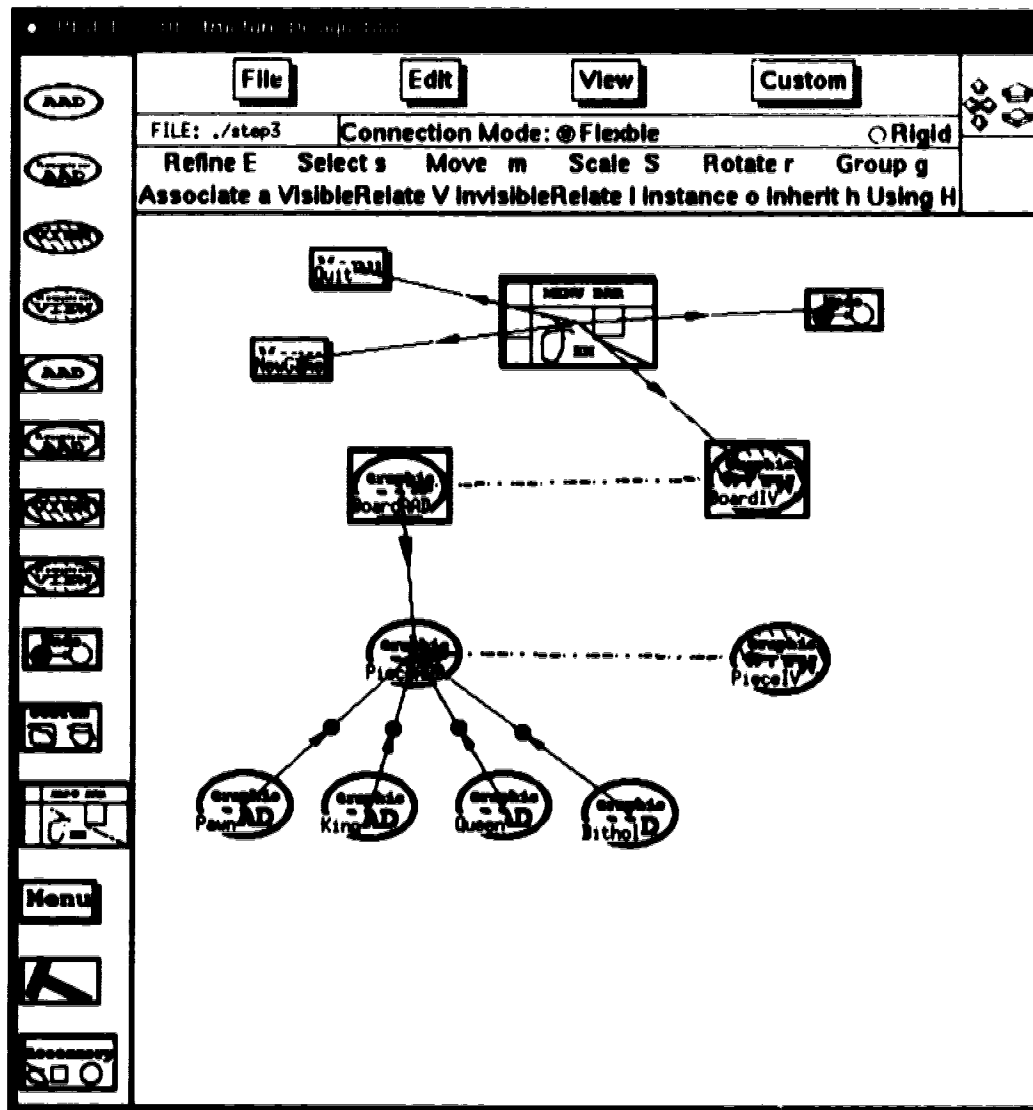


Figure 18: Building user interface structure

In particular object relationships can be used to derive the overall structure of the UI. In UISDT, these relationships among objects in the object model are the basis for the UI structure.

### **5.3 Design Process**

A fundamental design decision in UISDT is to support object-oriented methodologies, and UISDT offers designers a graphical editing environment to build object models [Boo91, RBP<sup>+</sup>91] of their applications through a direct-manipulation UI. An object model captures the static structure of a system by showing the objects in the system, the relationships between objects, and the attributes and operations that characterize each class of objects.

The design process of an OO interactive system can be described as a modeling process [Boo91, RBP<sup>+</sup>91]. The modeling process takes place between the real world (the referent system) and a software model (the model system). The reference system consists of phenomena that have definitive existence in reality or in the mind. To build the model system, the designer finds elements that model the phenomena and objects in the OO paradigm that are natural models of phenomena. To simulate the real world, behaviors of objects have to be defined, which leads to the specification of relationships among the objects.

There are four aspects of the OO design cycle, namely abstraction, interface design, implementation and verification. The primary step of OO design is to identify the objects or candidate abstractions of the problem domain and their relationships. An object explicitly embodies an abstraction that is meaningful to its clients. Abstractions can be generally defined as the identification of concepts in a domain such that similarities are fundamental and differences are insignificant for the purpose at hand. The abstraction is characterized by a set of services (operations) that clients can request. The relationships between objects determine the ways the objects work together to meet the requirements of a particular application. The interface design phase translates abstractions into a computer-specific form, specifying class definitions, class hierarchy and class behavior, i.e., defines classes of these objects and structures the system. The implementation phase is the point at which implementations are provided for the class interfaces designed in the interface design phase. Finally, the validation phase checks that the resulting system adequately models the reference system and meets other requirements. This validation of the model system leads

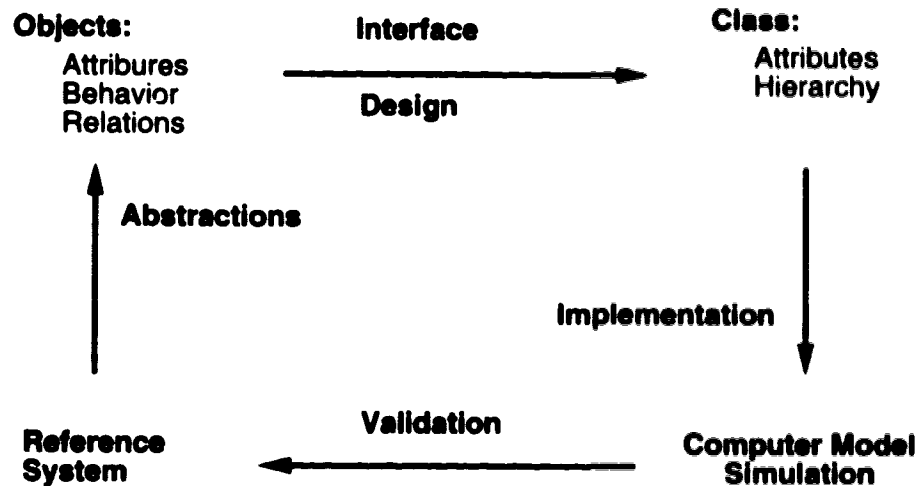


Figure 19: Object-oriented software development cycle

to the start of the next design refinement cycle. Figure 19 diagrams a typical development cycle for constructing object-oriented software.

UISDT is designed to support the above OO design process cycle, particularly the last three phases.

### 5.3.1 Component Definition

Defining components in an object model is the most important part of a system design. The designer expresses the domain concepts in terms of objects existing in the application. The questions that the designer keeps in mind are: what are the different kinds of entities that my program needs to manipulate? and how do these entities interact? The design process is simplified in UISDT by building an AAD tree and IV tree(s) which represent the object-model of the application. Domain concepts are expressed by UI objects, which in turn split functionality between the AADs and IVs. UISDT provides the following components as the basis on which an application-specific object model can be built:

**AAD** defines the base class for objects that store, provide access to, or modify the abstract information of the application that is directly related to user interactions. AADs are

used to define domain-specific interaction semantics. There are general ADDs and GraphicAADs (for structured graphics); primitive AADs and composite AADs.

**IV** defines the base class for objects that present, or allow the users to directly manipulate, the information stored in its corresponding AAD. IVs specify domain-specific interaction behaviors. Both AADs and IVs are referred to as domain objects that define and manage information to mimic the behavior of real-world objects. There are general graphical IVs (for displaying structured graphics); primitive and composite IVs.

**UIShell** is the root of the IV tree that has user-accessible commands and tools and provides multiple views of the same or different IV sub-trees. UIShell defines operations to initialize the environment, interpret application parameters, and run an event dispatch loop.

**User-Transparent Objects** define application objects that are not directly related to interaction, but contain the functional core of the system. The user-transparent components help integrate the UI and the application code, and their relationships with UI components keep the two sides consistent amidst changes to one or the other.

**Command** components map some of the external attributes and methods of an object model onto a set of controls, such as menus and dialogs, that are common control objects in many GUIs. Examples include a menu item for selecting and executing an action on a object, or a dialog box for changing attributes of domain objects.

**Tool** components present interactive features of domain concepts to the user. Examples includes tools for selecting domain objects for subsequent editing and for connecting domain objects. Tools are different from commands in that they support the direct-manipulation of domain objects. The concepts of command and tool are similar to those in Unidraw, and menus and tools in Macintosh applications.

**User-support objects** provide services for other domain objects in the system to implement user support facilities. UISDT provides two kinds of user support objects: undo and customization. The designer can extend their default behavior to specify domain-specific semantics.

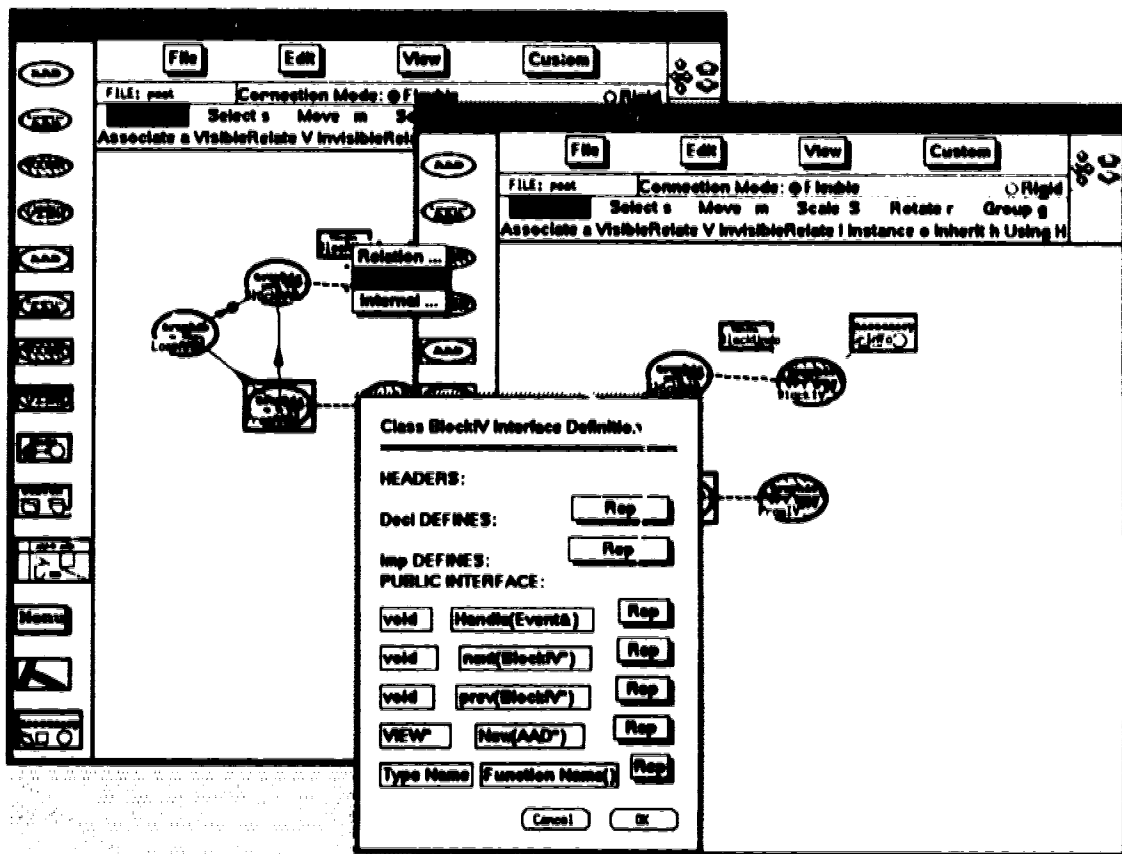


Figure 20: Define Component

**Accessories** are the components that supply UI details for control and decoration. Examples include a scrollbar for controlling the view of a large domain object (like text or graphics) and an active message for displaying status information.

Figure 20 shows a component defined in UISDT: the user is inspecting the behavior of the **BlockIV** component in the POAT structure whose design will be presented in the next chapter. The designer can instantiate any of these components by first engaging the component tool and then clicking in the viewing area. The designer can then inspect, modify, or define new application-specific behaviors and properties of the component using the refining tool. After engaging the refine tool, the designer can examine the **BlockIV** component by clicking on it. A menu pops up with three items: Relation, Behavior, and Internal. By choosing the Behavior item as in Figure 20, the designer views the behavior of the **BlockIV**. Now, the designer can modify methods by typing in the filed editor (return



type or function name field) or clicking the Rep button which launches a text editor to edit the code for that function. The designer can also add or delete a method.

There are three different aspects of a component that can be extended from the default behavior defined by UISDT abstractions: relationship with other components, external behavior visible to other components, and the internal implementation. The relationships between components are a critical part of the software structure and will be discussed in the next section. The external behavior defines the operations that can be applied to the component by other components. Each kind of component has default behaviors that can be overridden or extended by domain specific semantics in C++ style. UISDT supplies a behavior inspection and modification dialog box together with emacs-like text editor for the behavior specification. The internal implementation defines attributes and behavior implementation details which can also be inspected, modified, and extended in the same way as the external behavior.

### 5.3.2 Relationship Specification

Relationships are associations among the objects that define the structure of the system, whereby objects work together to provide system behaviors. These relationships are important in UISDT not only for describing the UI structure, but also for generating code in the prototyping stage. There are four important associations supported explicitly in the UISDT environment:

**IV-AAD (—) association** defines the connections between IV and AAD components and keeps consistency between them during interactions. The pre-defined protocols that implement this connection address the issues of what information IV(s) and AAD need about each other and how to get this information. This connection is the most important relationship in terms of UI structure.

**Containing (—) association** is the “part-whole” relationship in which objects representing components of something are associated with an object representing the entire assembly. For example, an IV can contain several other IVs as its components as in the chess example the board’IV contains piece’s IVs.

**Using (—) association** defines the link between two objects when one of them uses the services provided by another to perform its own action. For example, a command component would invoke a domain object’s method to respond to a user’s request.

**Inheritance** ( $\rightarrow\bullet$ ) association is the “is-a” relationship between a component and one or more refined versions of it, such as a default IV and an application-specific IV. Inheritance is helpful during implementation as a vehicle for reuse.

Relation tools in UISDT provide a direct-manipulation UI for establishing relationships between components. A relation tool involves two components, the source and the target. The designer identifies the source by downclicking on it, then moves the mouse with a rubberband line to the target, and finally selects the target by upclicking on it. UISDT will interpret the connection according to the context and establish the appropriate connection between them, i.e. the semantics of the connection depends on the components involved. For example, relating an AAD and an IV establishes an IV-AAD association, while connecting two AADs establishes a containing relationship. Relating a command (or menu) component to a UShell means that this UShell component contains this menu component in its layout, while associating a menu component to an IV implies that when the user selects this IV, this menu will be popped up. To avoid cluttering the display and distracting the designer by having too many connection lines, some of the associations such as the using relationship are not visible through connecting lines (unless requested by the designer), but displayed in their relation inspection dialog box. The designers can also set up a connection between components through a relation inspection dialogue instead of through direct manipulation.

### Structure Editing

In UISDT’s viewing area, components in the scene (i.e. in the specification of a UI structure) can be moved, scaled, duplicated, added, and deleted. The associations can also be added and deleted like components. Components can be grouped as one geometrical object or ungrouped as several independent geometrical objects in the viewing area. The group operation has different meanings when it is applied to different kinds of components. For example, when a set of menu components are grouped, this grouping operation establishes a hierarchical relationship among the grouped components and this relationship leads to the generation of a menubar that contains a set of menu items.

In order to better support the graphical editing metaphor, UISDT uses both constraint and persistence techniques in UI structure editing. Constraint mechanisms are used to make it easy for the designer to place components in the viewing area and maintain the layout. The constraint technique is used to establishing and maintain the associations in the viewing area, i.e. managing the geometrical aspects of the components in the viewing area. The

components supported by UISDT are graphical objects. The designers attach constraints to these objects resulting in constrained lines, i.e., when two components are connected by a line, a geometrical constraint is established. Supported constraints include horizontal, vertical, equivalency (two points in an equivalency relationship always move to the identical location), fixed length, and slope. These are binary and bidirectional constraints: once a binary relation is imposed on two objects, moving either one forces the underlying constraint satisfaction system to reposition the other to satisfy the constraint.

The persistence technique is mainly used to save and restore the intermediate results of the ongoing design. Each component supported by UISDT has the ability to manage its state across sessions. That is, components can save as well as restore their specification information, geometrical information (including geometrical constraint if they have any), and information about their relationships with other components in a system.

### 5.3.3 User Support Facilities

The difficulty in building user support facilities in most UIMSs is due to the lack of a good framework. One unique feature in UISDT is that it provides a powerful Undo and customization frameworks that allow designers to build user support facilities into their UI structures when they define the UI architectures.

Our event-object undo model proposes a new way of recording recovery information and organizing the undo/redo facility. The *Undo* component provided by UISDT makes it very easy to build an application-specific recovery facility at the time the designer constructs the UI structure. The *Undo* component encapsulates the basic behavior of the local recovery facility described in chapter 4 and can be extended to have component-specific recovery semantics when the designer defines the domain components. By associating an *Undo* object to a domain object (IV or AAD) via the relation tool, the designer assigns the selected *Undo* object to provide recovery service for that domain object. By overriding the default Undo method in the undo component, the designer can define the component-specific undo facility. After completing the UI structure specification, the designer also installs an application-specific recovery facility that utilizes the UI structure for organizing local recovery facilities.

The same approach is employed by UISDT for installing the user customization facility. The *CusObj* components provided by UISDT define the customization support for major interactive components in UISDT, namely the IV, Command, and Tool components. The designer can extend these pre-defined customization components, particularly *IVCus*, to

have component-specific customization support. The designers are encouraged to define their own domain-specific models and integrate them into the modeling component instead of using pre-defined models.

## 5.4 Code Generation

A UI development tool's objective is to implement a UI from the user's high level specification. Though this prototype tool's goal is to assist the UI design, particularly structure construction, it also supports incremental, evolutionary development by generating structure definitions as well as toolkit code and performing test runs. By drawing system structure in the viewing area, the designer can quickly generate an initial prototype with simple behavior. This application-specific initial skeleton is the starting point for iterative UI design. UISDT also provides a persistence mechanism for its components' representation and their associations so that the designer can save the ongoing UI design across sessions and reload it later. This persistency makes it easy for iterative design and testing.

The code generation algorithm makes it possible to transform the specific architecture to the code structure of the final implementation using a toolkit. The approach we took to design the algorithm was to make the components of a structure independent of a particular toolkit, while the transformation mapped these generic components to a specific toolkit code. The algorithm makes use of structural relations (inheritance, composition, and using relationships) from the design and automatically generates all the structural information (i.e. relationships amongst components) and part of the behavior code. The components in UISDT define the protocol to specify their behavior and have very little information about window-level implementation. The algorithm maps these components to the components provided by a toolkit that implements interaction techniques.

Generation of a UI from an object model is not merely a matter of finding a formal mapping from component definitions of a UI structure specification to widget implementations in an available toolkit. The overall object structure and the relationships among the objects have an impact on transforming the specific architecture to the code structure of the final implementation using a toolkit.

We define a **CodeRep** as the base class from which the external representation objects for each component in the architecture are derived. The protocol of **CodeRep** specifies the algorithm that maps the structure component to the building blocks provided by a

toolkit and defines the necessary operations to generate the code representation for the component. Each subclass is responsible for generating the source code representation for the corresponding subject it represents. The various `CodeRep` classes, such as ones for IV, AAD, `UIShell`, etc, provide component-specific implementations of the code generation method.

The algorithm consists of the following steps:

- Analyze the component definitions, and build an index of component relationships.
- For every component defined, create its `CodeRep` instance.
- For every `CodeRep` object, link the `CodeRep` objects according to the corresponding component relations.
- Traverse the `CodeRep` hierarchy and invoke `CodeRep` object's methods for first generating class definition (header) files, then generating class implementation files.
- Build the application (generate the makefile and then make the application).

Code generation takes place in multiple phases guided by the relationships amongst components, particularly containing and using relationships, each traversing the entire `CodeRep` hierarchy. This hierarchy (i.e., the UI structure) mirrors the component hierarchy created by the designer. Each `CodeRep` object in the hierarchy generates the proper code fragment in a given phase and not all `CodeRep`s participate in all phases. This process is independent of any particular toolkit implementation. The current implementation of code generation maps IV objects onto the tools in the `InterViews` toolkit, a C++ object-oriented graphical UI toolkit [LVC89]. `InterViews` is a powerful toolkit that provides common UI objects, graphical objects, and layout objects with a natural C++ API.

The information collected in generating code for components in the UI structure is then used to generate the makefile for the application. The generated code and makefile can be used either by `UISDT` to build the application or by the designer to refine them by hand.

## 5.5 Discussion

In the development of the `UISDT` framework, an important goal was to design a generic architecture for interactive systems. An approach was proposed for decomposing these

systems, and supporting this decomposition with standard abstractions and hierarchical organizations. The architecture is specified in terms of the structure of the class and object hierarchies that would be expected in any interactive application. A “framework” of standard classes and mechanisms that would allow a prototype system to be assembled simply and quickly was implemented.

Peridot [Mye88] and UofA\* [SG91] attempted to produce such a generic architecture by following the *Seicheim* model starting with a functional decomposition of the application and the UI sub-systems. The UI is then further sub-divided into components such as presentation, dialogue and application linkage. However, their separations by functional decomposition breaks down when the interactive system is considered as a whole. The design of the application software is constrained by the requirements of interaction and the UI software must take the application semantics into account.

## Chapter 6

# Experimental Applications

We have built graphical interfaces for three different applications to evaluate both the architectural model and UISDT implementation. Our aim was to demonstrate that the model supports diverse domains, that the abstractions provided by the model are powerful and flexible enough to produce application-specific UI structures, that UISDT reduces development time significantly, and that the resulting applications are comparable in utility to their conventionally-developed counterparts.

In the following, we use three applications that we built as examples to give a more detailed description of the UISDT environment. The applications include a performance monitor for an experimental virtual reality environment at U of A, a chinese chess program, and an interface for Enterprise, an interactive graphical programming environment for distributed systems. These applications represent typical graphical applications and show many important interface features. Moreover, there is little overlap in their design goals and the application domains.

This chapter covers both the design and implementation of these applications using UISDT. We conclude this chapter by evaluating the generality and productivity of UISDT.

### 6.1 Performance Monitor

The performance monitor, called POAT (Performance Optimization and Analysis Tool), is a software tool that provides an object-oriented, direct manipulation GUI for monitoring the performance of application programs running in an experimental virtual reality environment. The experimental virtual reality environment consists of: 1) two graphics

workstations for real time stereoscopic image generation; 2) an (optional) workstation for position and orientation sensing; 3) several (optional) workstations for real-time simulation/computation; and a high-speed network connecting these workstations. The performance of an application in this environment is mainly determined by the rate at which the display is updated. It has been observed that low update rates are mainly due to poor distribution of the computational load and delays in data communications and synchronization. Performance analysis requires real-time monitoring of the time spent in the blocks that make up the program. Optimization can be achieved by re-arranging program blocks in a manner that enhances performance. Because there are certain temporal dependencies among different blocks (e.g. the user's head position/orientation must be acquired before hidden surface removal can be performed), the re-arrangement must satisfy a set of constraints based on these dependencies.

### **6.1.1 POAT's Functionality**

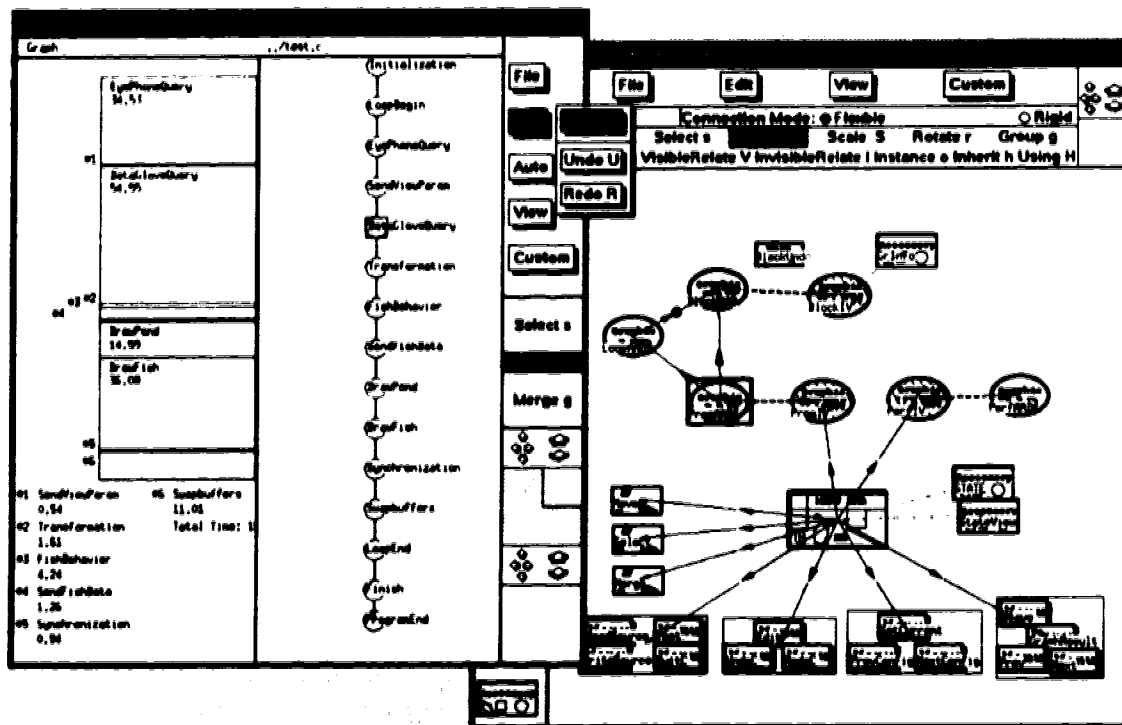
POAT is designed to allow a user to interactively analyze the performance of application programs, and to optimize their performance through automatic or manual changes to the program structure while observing certain constraints. It provides the following functions:

1. automatic decomposition of program code according to directives in the form of comments, and the graphical display of the structure,
2. the input of user specified constraints,
3. automatic insertion of timing code into the program according to the decomposition,
4. test run of the program in the virtual reality environment,
5. performance data gathering and graphing,
6. traversal of all possible alternative program structures satisfying the constraints, and
7. interactive modification of the program structure, with constraint checking.

In addition, due to the nature of multiprocessing for virtual reality applications, POAT is a multiple-view interface that monitors several programs running on several workstations.

Figure 21 shows a monitor session and depicts the default interface schematically. After POAT reads in the source code and its accompanying constraints, it displays the program





**Figure 21: POAT system prototype and structure design in UISDT**

structure on the screen. Each block of code is shown as a node on the diagram. The user can invoke a visual editor provided by POAT to modify the code within a block. This allows the user to further decompose the source code by adding more directives. Likewise, POAT provides a merge operation which allows the user to combine several blocks into one. The user can change the structure by dragging the nodes to the desired position in the diagram. If the change violates certain constraints, POAT will issue a warning message and undo the change. After the user feels ready to perform a test run, he can choose the code generation function to produce a version of the program containing timing code and then perform the test run. The performance data is gathered automatically by POAT and graphically displayed on the left portion of the monitor.

### 6.1.2 POAT's Architecture

The first step is to build the application model. In UISDT, an application model is a hierarchy of AADs that specify application processing and data structures; the UI for the

application model is a separate hierarchy of IVs that defines views and the user accessible control aspects of the application. Program blocks, program, and performance data are the domain objects with which a user interacts. We define **blockAAD** to model the program block concept, **programAAD** to represent a program that contains a set of **blockAADs**, and **perfAAD** for collecting, storing, and managing performance data. These components and their relationships represent the application data model. A **blockAAD** represents a program block. Associated with each **blockAAD** is a segment of code, a block name, and the geometrical shape used to represent the block graphically. The **blockAAD**, a subclass of the **GraphicAAD** component, includes the extended behavior for program block specific semantics, i.e. a set of properties and a set of operations to access and modify these properties. A **programAAD**, representing a program, is a composite object that contains a list of **blockAADs** as its components. The **programAAD** is derived from the composite **GraphicAAD** to represent the semantics of a program as well as its structure. When an instance of **programAAD** is created by reading a program, a set of **blockAADs** as well as their **blockIVs** are also created as a result. The **perfAAD**, a subclass of **AAD**, has the data structure to represent the performance data of a program's execution and provides a set of operations to manipulate this data structure. An instance of the **perfAAD** is used to collect and store the performance data for one particular configuration of a program. In POAT, a **programAAD** object can be associated with a set of **perfAADs** giving its performances in various configurations.

Next, the IVs required for the above AADs are defined and placed in the viewing area. **BlockIV**, **programIV**, and **perfIV** are defined to map the above domain concepts respectively onto the UI as interaction objects (see Figure 21). A **blockIV**, a subclass of **IV**, defines an interactive view for a **blockAAD** that allows a user to view as well as directly manipulate a program block graphically. A **programIV**, a subclass of composite **IV**, represents a **programAAD** on the display and provides the user with direct access to program structure. A **perfIV**, a subclass of **IV**, displays its **perfAAD**'s state (i.e. performance data stored) graphically on the UI, but no interaction is allowed.

Finally, the root of the IV tree is a **MonitorIV**, a subclass of **UIShell**, that represents a top-level view for POAT; provides menus to issue commands such as editing the code in the currently selected **blockAAD**, reading in a new program to create a **programAAD** and a **programIV**, performing a test-run, collecting and graphing performance data (creating **perfAADs** and associating these **perfAADs** with the **programAAD** on the display). Tools to manipulate **blockIVs** such as selecting, moving, and merging; labels to display status; and

dialog boxes for entering data and showing warning message are provided by the **MonitorIV**. The **master-monitorAAD** derived from **MonitorIV** maintains several **monitorIVs** that are started for monitoring slave programs running on different workstations.

After building the structure of **POAT**, we can invoke the **generate** command in **UISDT** to generate **C++** source code together with a **Makefile**. We can then build **POAT** and execute it. We can refine the system by repeating the above steps. The undo facility is useful in **POAT** allowing the user to reverse his/her changes on the program structure. By extending the default **RO** and **RI** objects to include recovery information (translation distance) and the reverse-translation operation and attaching the extended **RO** to **blockIV**, we can install undo facility in **POAT** easily.

## 6.2 Chinese Chess Program

An object-oriented, direct manipulation interface has been developed for a Chinese Chess Program called **Abyss** [Ye92] using **UISDT**. **XAbyss** is a chinese chess program developed at the University of Alberta that plays close to D class of chinese chess and has a direct-manipulation interface (see figure 22). **XAbyss** has been distributed freely on internet and has a substantial following of users.

Chinese chess has lots of similarities with chess, and therefore, the analysis we had for the chess program used in previous chapters can be applied to **XAbyss**. In this section, we only outline the system structure, instead of the detailed analysis of the design. Using our abstract model of interactive systems, the **XAbyss** is separated into three parts: user-transparent application objects which include most of **Abyss**, **UI** objects which define the interface between the user and the user-transparent application objects, and user-support objects which, through their cooperation with other objects, provide user support facilities for undo/redo and customization.

The system has four kinds of objects: **piece**, **board**, **monitor**, and **machine-player**. Each **piece** is an interactive graphical object which has a position on the board as its internal state. A **piece** can send messages to the **board** for notifying its change, to the **monitor** for verifying its movement, and to the **machine-player** for telling its new position. The **board** is a composite object that contains all **piece** objects as its components. The **monitor** and the **machine-player** objects are not visible to the user, but contain most of the

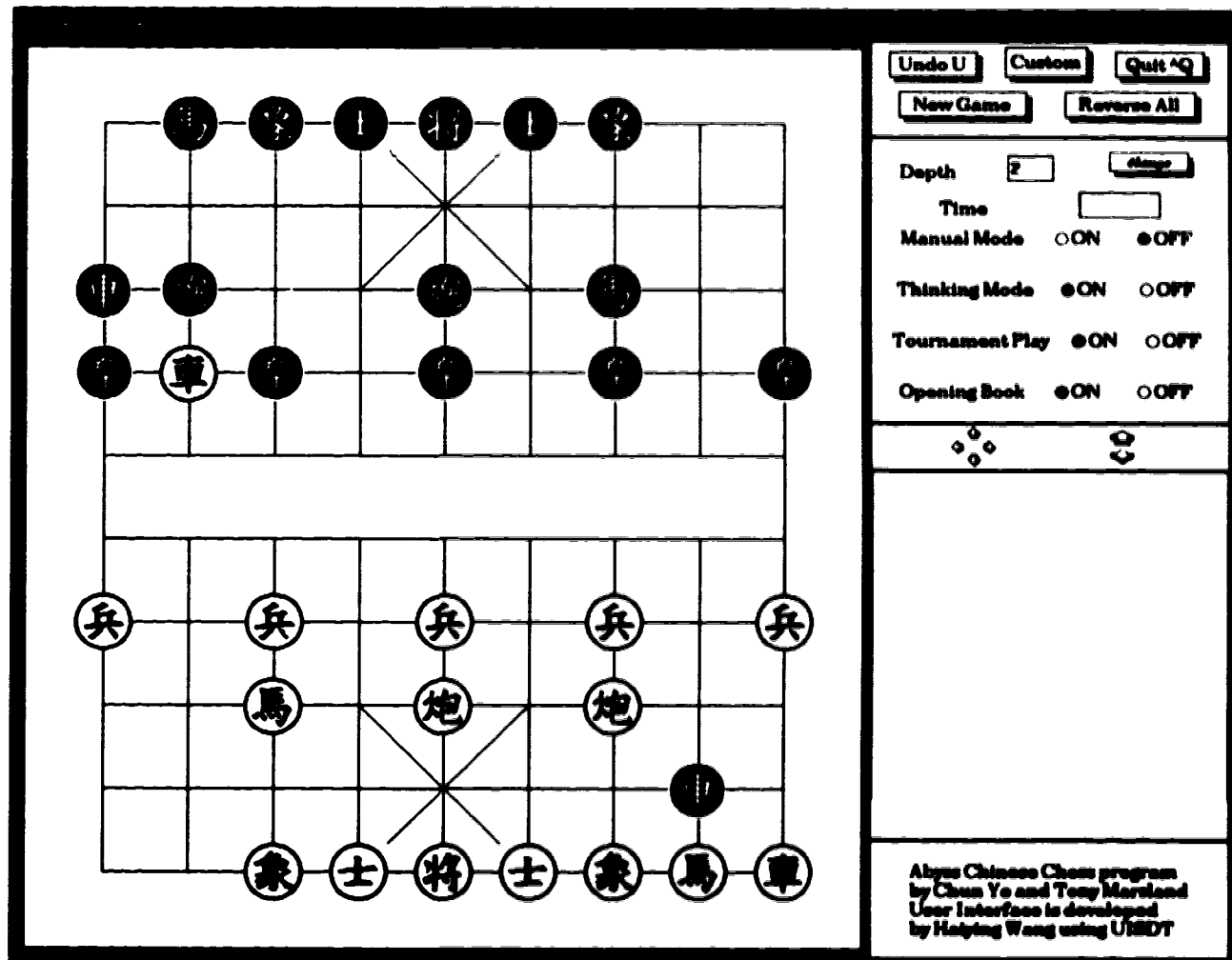


Figure 22: Chinese chess program

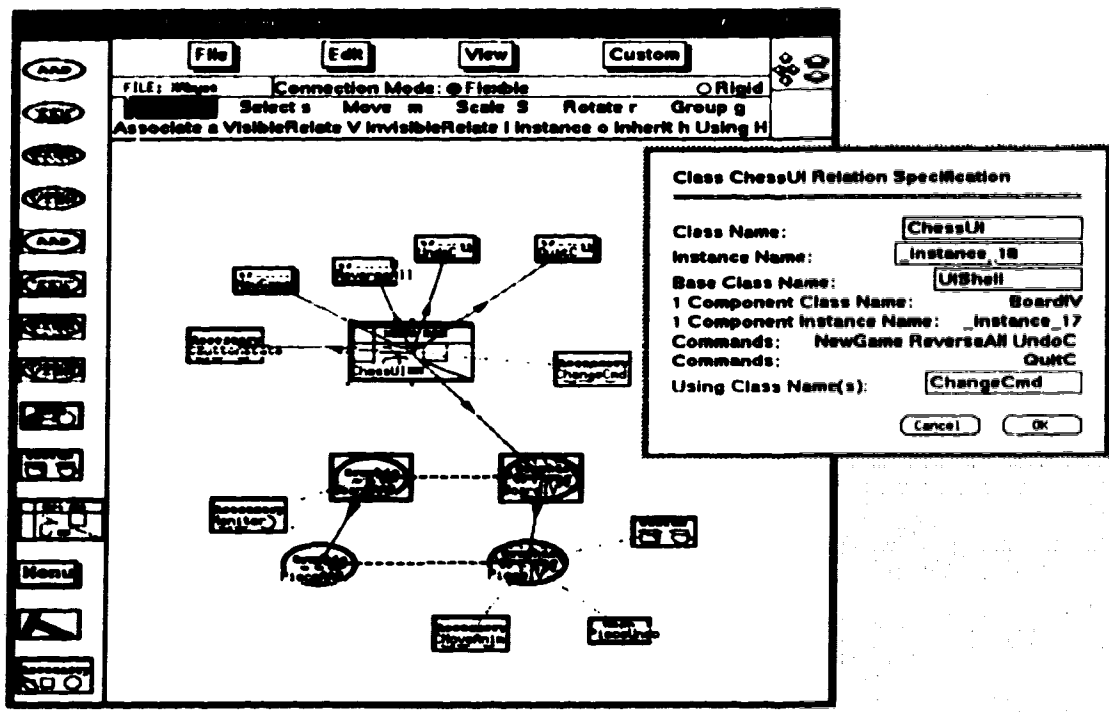


Figure 23: Chinese chess program structure in UISDT

application related knowledge. The **Monitor** and the **machine-player** are user-transparent application components.

The UI objects use an AAD-IV separation: the application data is structured as a hierarchy of AADs: **PieceAAD**, **BoardAAD**; the view structure is a hierarchy of IVs: **PieceIV** and **BoardIV**. The **ChessIV** is the top-level window for displaying the chinese chess program, providing buttons to control the playing, a panner to zoom the board, and dialog boxes to display various messages (see figure 23). The way to define these components and layout the XAbyss structure in UISDT is similar to what we have described for POAT and we do not go to that detail here again.

One of the major parts in designing XAbyss is to install undo and user customization in XAbyss. We extended the undo and the customization components to provide chess-specific undo and customization. The installation of the undo framework is quite easy, since each **piece** has its own RO and the board has an RO which serves as the parent of the pieces' RO. The piece-specific RO is bound with the **PieceAAD**, instead of the **PieceIV**. The reason is that XAbyss can have two displays so that two players can play XAbyss

across a network and thus, each **PieceAAD** is associated with two **PieceIVs**. A **PieceAAD** has all semantics related with the piece it models. The ROs and the RIs are instances of the RO and the RI classes in the POAT program. The only change that was made was to change the **Undo** method so that it can interact with the two non-visible objects and perform semantic operations correctly during recovery. This application structure is similar to (2) in Figure 13.

For the purpose of training and improving XAbyss, we often need to repeat some openings, middle games, and end games, or when the program loses a game we want to replay the player's moves to figure out the problem in the program. These requirements lead to the need of for a customization facility that allows XAbyss to record certain move patterns and play them back as many times as we want to study the program's behavior. For this reason, we installed the customization framework discussed in section 4.2 into XAbyss. A **PieceIV** has its own IVCus to record as well as playback a user's interaction on a piece object. The modification to IVCus is made so that IVCus can use the knowledge of the piece in replay. For example, if piece A was moved from position 1 to position 2 in recording and in replay, and a piece on the other side of play is in a position that can capture the piece in position 2. In this situation, we can not move piece A from position 1 to position 2 any more in replay and piece A has to move to another position by asking for help from the **machine-player**. The models that are extended in the modeling component include several skeletons of openings and middle games in chinese chess. The customization facilities, consist of an XAbyss-specific modeling component containing models, customization objects attached to **PieceIVs**, the logger, and simulation agents, can map the recorded player's moves in the logger to one of these models and create the simulation agent. That is, we can generalize the trace of pieces' movements to one of these skeletons to turn the trace into a useful game.

### 6.3 Enterprise User Interface

Enterprise is a programming environment for designing, coding, debugging, testing, monitoring, profiling and executing programs in a distributed hardware environment [LLM<sup>+</sup>92]. The Enterprise system is built with the following objectives:

- to provide a simple high-level mechanism for specifying parallelism that is independent of low-level synchronization and communication protocols, and

- to provide transparent access to heterogeneous computers, compilers, languages, networks, and operating systems,

### 6.3.1 Enterprise Concepts

The overall organization of a parallel or distributed program in Enterprise is similar to the organization of a sequential program. The user views an Enterprise program as a collection of modules. Each module consists of a single entry procedure that can be called by other modules and a collection of internal procedures that are called for sequential or distributed execution. There is an analogy between Enterprise programs and the structure of an organization. In general, an organization has various assets available to perform its tasks. For example, a large task could be divided into sub-tasks where various sub-tasks are given to different parts of the organization (individuals, departments, lines or divisions) to perform in parallel. In addition, an organization usually provides many standard services (like information storage and retrieval) that are available on demand to improve its functionality.

Enterprise supports roles corresponding to six different asset kinds:

**Enterprise** An enterprise is a single program. It is analogous to one organization.

**Individual** An individual contains no other asset. An individual is analogous to an individual person in an organization. When called, an individual executes its sequential code to completion. Therefore, any subsequent call to the same individual must wait until the previous call is finished. In general, an individual can be replaced by a line, department or division at any time. However, there are two special kinds of individuals: one is called a **receptionist** and the other is called a **representative**. Receptionists serve as the first element of any composite asset and cannot be replaced by any other asset nor can they be replicated. Representatives can be replicated but they can only be replaced by divisions.

**Line** A line contains a fixed number of heterogeneous assets in a fixed order. Each asset contains a call to the next asset in the line. A line is analogous to a construction, manufacturing or assembly line in an organization where at each point in the line, the work of the previous asset is refined.

**Department** A department contains a fixed number of heterogeneous assets. A single receptionist asset shares its name with the department so that it can be called by

external assets. However, unlike a line, the other assets in a department do not call each other in a fixed sequential order. Instead, all other assets are called directly by the receptionist. A department is analogous to a department in an organization where a receptionist is responsible for directing all incoming communications to the appropriate place. A department consists of a collection of assets of any kind: individuals, departments, lines and divisions.

**Division** A division contains a hierarchical collection of identical assets where work is divided and distributed at each level. They can be used to parallelize divide-and-conquer computations. When a division is created, it has a single receptionist asset that shares its name with the division so that it can be called by external assets. In addition it has a single representative asset that represents the recursive call made by the receptionist to the division itself. The user may change the breadth of the division's first level by replicating the representative. The user may add a level to the depth of the recursion by replacing the representative by a division.

**Service** A service contains no other assets. However, unlike an individual that can only answer a single call at any one time, a service may be used by more than one asset at the same time. A service is analogous to any asset in an organization that is not consumed by use and whose order of use is not significant.

### 6.3.2 User Interface Functionality

In Enterprise, the user specifies the desired technique at a high level by manipulating icons using a GUI. Using the GUI, the user draws a diagram of the parallel computation and writes sequential code that is devoid of any parallel constructs. Based on the user's diagram, Enterprise automatically inserts all the necessary code for controlling the parallelism, communication, synchronization and fault tolerance. It then compiles the routines, dynamically assigns processes to processors and establishes the necessary connections. Processes run in the background, taking advantage of any available resources on the network.

Enterprise's UI was designed to allow a user to express parallelism in a simple graphical editing manner. In Enterprise, the application graph is an asset graph and it is constructed in a novel way. The user starts with an asset and constructs the graph by replacing and expanding individual icons corresponding to six different assets described previously above. Figure 24 shows Enterprise's UI.



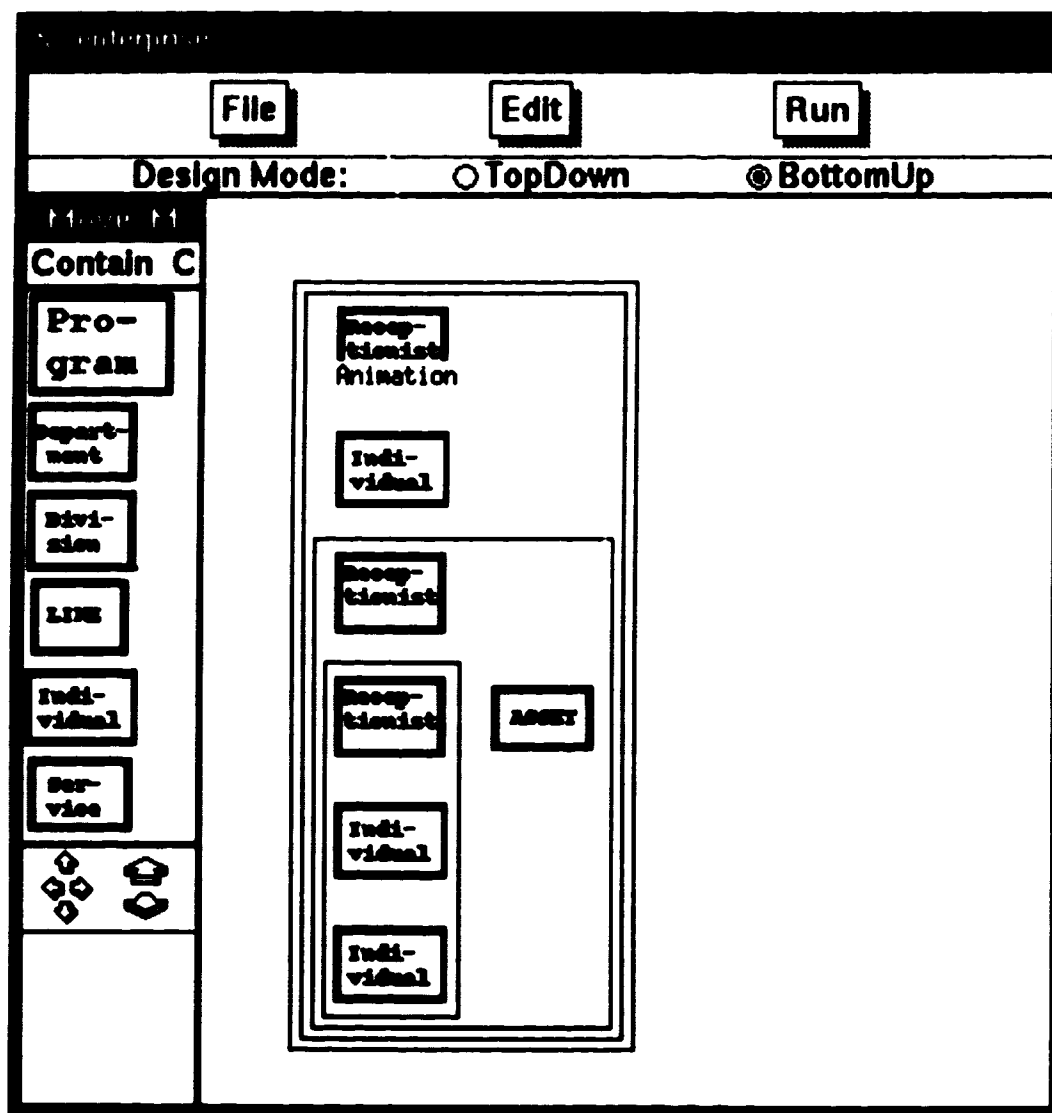


Figure 24: Enterprise at work

Each asset has a context-sensitive menu. If the user presses the middle mouse button when the cursor is over an asset, then a pop-up menu appears containing all of the operations that are valid for that asset. The user can also invoke the operation through menu bar. The following operations can be performed on assets, although not all are valid for all assets:

- name or re-name the asset,
- open an edit window on the code of the asset,
- read in a file containing the code of for the asset,
- write the code of the asset to a file,
- replicate an asset by providing minimum and maximum replication factors,
- replace an asset by a **Department**, **Division**, **Individual** or **Line asset**,
- add or delete an asset,
- expand an asset so its component assets are displayed, and
- collapse an asset so that its component asset are hidden.

In addition, the following operations can be performed on the viewing canvas in general:

- a) create a new program consisting of a single enterprise asset and display it on the canvas,
- b) save the current program,
- c) input an existing program by offering the user a file chooser and displaying its graph on the canvas,
- d) compile the current program,
- e) run the current program, and
- f) undo or redo the user operations.

### 6.3.3 User Interface Implementation

Figure 25 shows the object-model of the Enterprise user interface designed in UISDT. It describes the system structure together with its abstractions. The class organization among **Asset** classes describes their behavior and relationships. **Asset** class is an abstract base class for asset components that defines common behaviors and basic protocols for asset objects, such as naming, reading and writing the code, and editing the code, and is connected with the **AssetIV** class which specifies interactive behavior of asset objects. Components, such as **Service**, **Individual**, **Receptionist** and **representative**, are inherited from **Asset** directly. In addition to the behavior of **Asset** class, these components define and implement their semantics-specific properties and operations described before.

Notice that the **AssetIV** class is connected with a set of menu buttons. In UISDT, the grouping operation over a set of menu items means that these menu items are organized as a menubar and the association between a menu (menu item or menubar) and an IV component means that the menu is a popup menu associated with that instance of the IV and the menu will be popped up whenever a user manipulates the IV component (default operation is to press mouse button 2). Although nine operations are defined in the menubar, not all of them are active to all **Asset** instances. For example, *Expand* and *Collapse* does not make sense for primitive **Assets**, such as instances of **Service** and **Individual**. *Replace* is not allowed to apply to instances of both **Representative** and **Receptionist** and neither is *Replicate* for instances of **Receptionist**.

The **Compose** class is an abstract subclass of both the **Asset** and composite AAD that adds some composition behavior to the **Asset** class so they can contain other **Asset** instances as their components. Components, such as **Line**, **Dept Division** and **Program**, are subclasses of the **Compose** class with specific behaviors. The **ComposeIV** class is a subclass of **AssetIV** that provides interactive view for **Compose**. All the nine menu items in the popup menubar are active under instances of **ComposeIV**, though the context and semantics may not be the same. *Expand* and *Collapse* operations allow a user to view an **Asset** as well as the program at different levels of abstraction.

**Workspace** is a subclass class of composite AAD which organizes **Asset** components and conveys them to the user through **WorkspaceIV**, a subclass of **Viewer** described in section 3.3.4. The reason for introducing **Workspace** and **WorkspaceIV** is to support the selection concept so the system menus as well as the tools discussed later can be applied to a selected **Asset**.

**UI**, derived from **UIShell**, is the top view of the Enterprise UI. It contains **WorkSpaceIV** as its viewing area for a user to design parallel programs and a set of commands as well as tools for helping him manipulate and control his design. The system menus, such as **Input**, **Save**, etc., are derived from **Command**, with application-specific extensions. There are two kinds of **Tools** associated with **UI**: **Tools** to set the manipulation style, such as *Move* and *Contain*, and **Tools** to instantiate instances of **Asset**. By selecting *Move* tool, a user can move **Assets** around inside **WorkSpace**, and by selecting *Contain* tool, a user can let one **Asset** contain another **Asset** as its component. **AssetTool** is an abstract base class for asset component tools that allows the user to instantiate instances of various asset components in workspace.

## 6.4 Experience With UISDT

Building **POAT**, **XAbbyss**, and the Enterprise UI with **UISDT** demonstrates how the architectural model facilitates the design of UI structures and how the tool supports UI implementation by transforming the architecture into the final software code. Each application has features that set it apart from the others, thereby offering different perspectives on the architectural model. This section discusses the generality as well productivity of **UISDT** based on the experience of producing the above three applications.

### 6.4.1 Generality

It is very difficult to discuss formally the range of UIs that **UISDT** can create because there are no taxonomies of existing UIs. **UISDT** was designed to support graphical and direct manipulation UIs in general with a pointer device, bitmap display, and an event-driven control structure. The three applications are typical examples of direct-manipulation GUIs with manipulable graphical objects and visual controls, such as menus and tools. Theoretically, almost any direct-manipulation GUIs could be build with the **UISDT** framework. The **UISDT** framework provides all the necessary abstractions a designer can use or extend to build GUI-based applications. The three applications are different enough to preclude easily turning one into another. For example, **POAT** is designed to have sophisticated background computations, and **XAbbyss** is meant to facilitate the design of parallel programs. However, the current implementation of the framework limits the possible interfaces and platforms. Additionally, the three applications, though their application domains are very different,

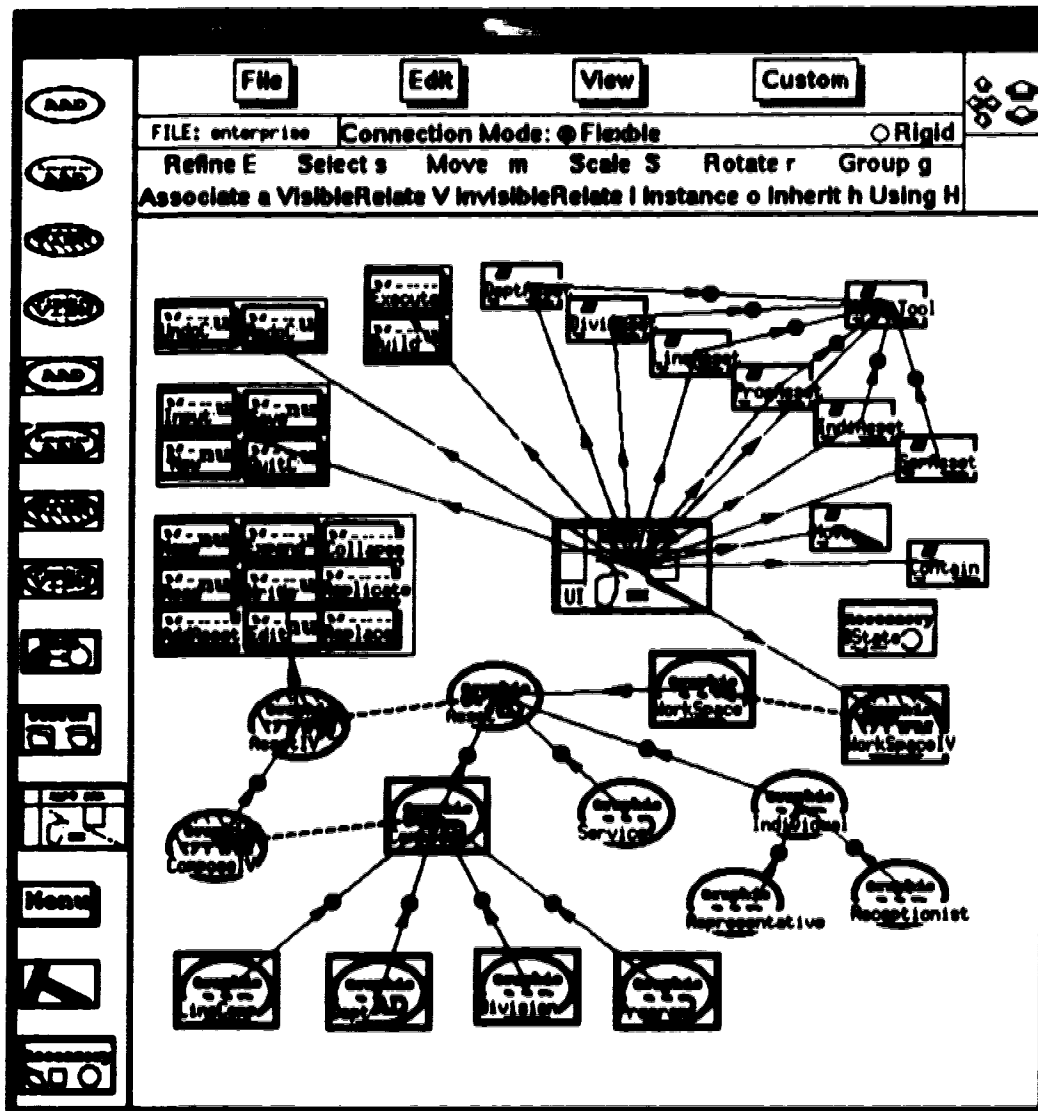


Figure 25: Enterprise user interface structure in UISDT

have many similarities in their UIs: they all are graphics-based front-end GUIs with simple application data structures. It would be more convincing if we had built a wider range of applications. For example, a GUI for a database application and a GUI for text editing. The reason for selecting the above three applications is largely the range of applications available at the time the author was looking for applications to test UISDT. Finding a good application itself is a challenge for GUI research.

### 6.4.2 Productivity

One of initial goals in building a UI development tool was to increase the productivity of constructing UIs. The experience of using UISDT has shown that UISDT has achieved this goal through its high level abstractions and prototyping facilities. In addition, the power as well as the generality provided by InterViews, especially its high-level encapsulation and its composition abstractions, makes the mapping from the UISDT abstractions (IVs) to InterViews classes easy and flexible. Using UISDT, the designer can easily create a structure design for an application and then implement or modify it. With the quick prototyping ability, the designer can minimize the effort required to iteratively refine the previous design in the UISDT environment. Each of these three applications built with UISDT reflects a significant reduction in development effort compared with designing from scratch using GUI toolkits directly.

Of the three experimental applications we built with UISDT, the Enterprise UI gives us the best opportunity for comparison with existing prototypes. There were three different UI designs and implementations before UISDT was used to build Enterprise UI:

1. an object-oriented design with C++ implementation using InterViews toolkit, its development took about a half year from the design to prototype,
2. a different design with almost the same functionality as 1) with a C implementation using Xt toolkit, it took about four months to complete this prototype, and
3. an object-oriented redesign with a Smalltalk implementation, this last effort took about one month from design to prototype.

Without sharing the design knowledge as well as the implementation experience of the above development efforts, the author took only one week to complete the design and implementation of the new Enterprise UI using UISDT. This prototype provides approximately the same functionality as the previous prototypes.

Note that UISDT is the result of an individual research project and an experimental tool in the testing stage. It has been used so far mainly by the author. The claims made about development time reduction is primarily based on a) observations of how UISDT supports and facilitates the development process of UI design and b) comparisons with the author's experience of writing UIs with and without UISDT. The claims would be more convincing if we had let other UI developers build UIs with UISDT. Due to the time constraints, we were unable to perform such an experiment.

## Chapter 7

# Conclusion

We began this work by examining an important aspect of user interface development, UI structure design. This aspect has been difficult to design, and the least supported part of current UI development tools, despite advances in UI technology. This problem prompted our hypothesis that a UI architectural model need not support the construction of all other higher level components other than the basic interactive and application components, as well as providing a means for combining these components to build up application specific components. This hypothesis in turn led to an experiment in which we formulated such a model, an application-oriented UI architectural model, extended this model to embed user support frameworks, implemented the model by building a library, and tested the model by constructing a UI design tool to verify our thesis.

### 7.1 Summary of Work and Contributions

This research contributes to user interface development by achieving the following goals.

**An application-oriented user interface model.** The model identifies the designer's needs by defining a set of components and provides mechanisms that allows the designer to compose these components to meet application needs. The model greatly facilitates the design of UI structures, bridging the gap between the domain concepts and UI components, and narrowing the design space of interactive systems. We believe that, by defining the basic components on which application-specific structures depend, the model can support a greater range of applications without pre-judging their structures.



**User support facilities.** The holistic view of UI structure we took gave rise to research on formulating and testing user support features in UI structures. These features are treated as structure properties rather than the properties of an individual UI objects.

- Our recovery approach of dividing traditional history/command lists into per-object lists is different from the other recovery methods in that it meshes well with object-oriented structure. It allows recovery to be handled without violating object-oriented principles or restricting the structure of object-oriented systems.
- Our domain independent object-oriented framework for supporting programming-by-example in UIs simplifies the construction of domain-specific customizations by providing programming abstractions that are common across domains.

We believe that interaction history is better collected at the relevant level, which in object-oriented systems is at the object level. User support concepts and functionality must pervade throughout the system, and the structure of the support facilities should be flexible enough to reflect the different interaction patterns of application objects. The Undo and the customization frameworks have simple semantics and fit well within the UI architecture. The abstractions provided by the frameworks make it easy to install user support facilities in UIs systematically.

**A non-trivial user interface development tool.** UISDT is a UI tool based on the architectural model that is special for UI structure design. UISDT provides an environment for connecting up the various parts of the architectural model. UISDT's implementation simplifies UI structure design by providing high-level abstraction components and reducing the effort in structuring a UI. UISDT has proved to be flexible and powerful enough to construct application-specific UI structures, and eliminates the need to hand-craft most of the code for specifying UI structures. UISDT assists in settling many details of structure design before the designer starts. A potential drawback of this approach is that it may place restrictions on what the application can do; however, we did not find this to be a problem when we built our experimental applications. The granularity of the components lets us do what we want and the result has reassured us that the architectural model is sound.

## 7.2 Future Work

The work reported here can be extended and improved in many ways.

### 7.2.1 Productivity Measurement

The goal of UI development tools is to increase the productivity of UI construction. We have argued that the proposed architecture model is better than the models used in UIMSs and toolkits in terms UI structure design. Experience with using UISDT to build applications has supported this argument. However, a more comprehensive and systematic measurement is needed to support this argument. This measurement should take various aspects of a UI design as well as different application domains into consideration. The experimental applications should be constructed separately using a UI builder, a UIMS, and UISDT that have similar functionalities. The goal of this measurement is not only to further back the claims made by this thesis, but also to set up a benchmark for similar comparisons. Success of the experiment depends on the selection of the applications and the criteria for the comparison.

### 7.2.2 New Features in the Model

The proposed architectural model is an attempt to be an application-oriented model by identifying what the designer needs and providing abstractions to address these needs. The current abstractions address only the common UI aspects in graphical UIs as shown in our experimental applications. We would like to go beyond the current limited support to develop a more powerful model, that is, not just traditional graphical UIs, but also CSCW or multimedia UIs. The possible extensions to the model in terms of features include, but are not limited to the following:

**Three dimensional user interfaces.** Many design and implementation decisions in the model were made without considering three dimensional graphics. Future refinement of the model is needed to support these features. This will lead to the extension of the IV protocols since they are the components dealing with display.

**Constraint system** Constraint mechanisms are an important paradigm in user interfaces, allowing the designer to specify interactions among components in terms of their relationships. We can add a constraint system to the architectural model by extending

the protocols of the components. For example, a simple geometrical constraint system that can be used to define the geometrical relationships between interactive components to specify the screen layout. In this case, the constraints (or relations) can be defined among IVs and the constraints would be notified in a fashion similar to the way the IVs are notified when an AAD changes state.

**Multimedia GUIs** Multimedia GUIs have become popular among modern UIs with the advance of sophisticated interactive devices. We can add multimedia features to the existing architectural model by extending the IV components. The protocols of the existing IVs are not restricted to graphics, although the current implementation is on top of a window system. Of course, by introducing new media into IVs, there are many other problems, such as those related to storage, retrieval, composition, and synchronization. New protocols and composition mechanisms are needed to integrate multimedia features into the model.

### 7.2.3 Component Refinement

The components of the model are undergoing continuous refinement as we use them to build different applications and examine various features of UIs. The emphasis on application-orientation needs more effort in identifying the tasks the UI designers face in building application-specific UI structures and to providing better abstractions to support the tasks.

The design of the prototype library is guided by the principle of separating the interface objects that define the abstractions of the UI model components from the implementation objects that realize the interface protocol by utilizing an available toolkit. Our current implementation achieves this by having two different types of classes: interface classes that are related to the abstractions of the model and independent from the specific details of the implementation, and implementation classes that are objects in an available toolkit and hidden from the user as a member variable of the interface class. This method makes it difficult for the code generation in UISDT to take advantage of the library implementation. [Gui91] uses multiple inheritance to combine two classes and we are interested in applying this technique in our library implementation.

#### **7.2.4 UISDT Extension**

The interactive display organization and layout of UI components is supported by the knowledge encapsulated in UISDT. Difficulties arise when a designer tries to change the default look-and-feel of the UI, since he/she must go through many iterations. Although the external appearance of a UI is not the focus in UISDT, this support is necessary for UI design. We are planning to take advantage of existing UI builders such as IBuild [VT91] to deal with the external appearance of the UI. These UI builders are powerful in terms of specifying the properties of widgets and defining their geometrical relationships. We will let UISDT generate the external representation of the UI in the format that other UI builders can understand and then refine the layout of the UI in builders.

Another possible extension to the current UISDT implementation is to have UISDT generate code for different GUI toolkits. Our code generation algorithm is not tied to a particular toolkit, though UISDT now can only produce InterViews based code. We can provide drivers for different GUI toolkits and by choosing a different driver, UISDT can generate a different toolkit-based implementation of a UI design.

#### **7.2.5 Powerful User Support Facilities**

In the undo framework, we have the following future work under consideration. First, we have a strong motivation to apply the recovery framework to various applications, specifically CSCW and multimedia user interfaces. The distributed and the object-oriented nature of the framework makes it suitable for both CSCW and multimedia application. In the current implementation, the framework supports only traditional GUIs. By providing more sophisticated ROs, such as ones for multimedia objects and groupware widgets, the framework would be able to handle various recovery operations in CSCW and multimedia applications. Second, the current framework proposes a new implementation, not a new semantics for the recovery. It focuses on organization and control of the recovery facilities in user interfaces. The investigation of new recovery semantics is important, especially in UIs that use state-of-art technologies, such as multimedia, virtual reality, and CSCW. We are planning to extend the applicability of the framework by incorporating not only more powerful ROs, but also new mechanism to handle new recovery semantics.

The customization framework has achieved its initial goal by proving the concepts of our approach, however, it has limited power in terms of building useful customization facilities.

It just defines a framework that an application-specific user customization can be built on. The hard problem in customization is generalization, and the algorithm that is used by the modeling component is the key for constructing a useful customization facility. In the future, we would like to investigate the following areas: more customization objects that could incorporate various interaction techniques and expand the generalizations that the framework is capable of making. We are looking for new applications that have well-defined domain so that we can define powerful models and generalization algorithms.

# Bibliography

- [And83] J. R. Anderson. *The Architecture of Cognition*. Harvard University Press, 1983.
- [And88] J. Palay Andrew et al. The andrew toolkit: An overview. In *Proceedings of 1988 Winter USENIX Technical Conference*, pages 9–21, 1988.
- [Arc84] J. E. Archer, Jr et al. User recovery and reversal in interactive systems. *ACM Trans. Programming Language and Systems*, 6(1):1–19, 1984.
- [Bar86] Paul S. Barth. An object-oriented approach to graphical interfaces. *ACM Trans on Graphics*, 5(2), 1986.
- [Bas88] Len Bass et al. Introduction to the serpent user interface management system. Technical report, Software Engineering Institute, Carnegie-Mellon University, 1988.
- [BC91] Len Bass and Joelle Coutaz. *Developing Software for the User Interface*. Addison-Wesley, 1991.
- [BF82] T. Blesser and J. Foley. Towards specifying and evaluating the human factors of user-computer systems. In *Proc. Human Factors Comput. Syst.*, 1982.
- [BKT82] H.G. Borufka, H.W. Kuhlmann, and P.J.W. TEN Hagen. Dialogue cells: A method for defining interactions. *IEEE Trans. CG&A*, 1982.
- [Bod91] Susanne Bodker. *Through the Interface - A Human Activity Approach to User Interface Design*. Lawrence Erlbaum Associates, 1991.
- [Boo91] Grady Booch. *Object-Oriented Design with Applications*. The Benjamin/Cummings, 1991.

- [Bor86] A. H. Borning. Constraint-based tools for building user interfaces. *ACM Trans. on Graphics*, 5(4), 1986.
- [Bux83] W. Buxton et al. Towards a comprehensive user interface management system. In *SIGGRAPH'83*, 1983.
- [CMN83] S. Card, T. Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.
- [Coo89] S. Cook, editor. *Architecture Models for Interactive Software*. Cambridge University Press, July 1989.
- [Cyp91] Allen Cypher. Eager: Programming repetitive tasks by example. In *SIGCHI'91*, pages 33–39, 1991.
- [dBFM92a] D. J. M. J. de Baar, J. D. Foley, and K. E. Mullet. Coupling application design and user interface design. In *ACM CHI'92*, 1992.
- [dBFM92b] Dennis J.M. de Baar, James D. Foley, and Kevin E. Mullet. Coupling application design and user interface design. In *ACM CHI'92 Proceedings*, 1992.
- [Doy79] J. Doyle. A Truth Maintenance System. *Artificial Intelligence*, 12:231–272, 1979.
- [DP87] R. Dechter and J. Pearl. Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence*, 34(1):1–38, 1987.
- [FB87] M.A. Flecchia and R.D. Bergeron. Specifying complex dialogs in algae. In *CHI+GI'87*, 1987.
- [FDFH90] J.D. Foley, A. Van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics - Principles and Practice*. Addison-Wesley, 1990.
- [FGKK88] J. Foley, C. Gibbs, W.C. Kim, and S. Kovacevic. A knowledge-based user interface management system. In *ACM CHI'88*, 1988.
- [Fou89] Open Software Foundation. *OSF/Motif Programmer's Reference, Revision 1.0*. Open Software Foundation, 1989.
- [Geo87] Michael P. Georgeff. Planning. *Ann. Rev. Comput. Sci.*, 1987.

- [GR83] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.
- [Gre85a] M. Green. Report on dialogue-specification tools. In G.E. Pfaff, editor, *User Interface Management Systems*. Springer-Verlag, 1985.
- [Gre85b] M. Green. The university of alberta user interface management system. In *ACM SIGGRAPH'85 Proceedings*, 1985.
- [Gre86] M. Green. A survey of three dialogue models. *ACM Trans. Graph.*, 5(3), 1986.
- [Gui91] Nuno Guimaraes. Building generic user interface tools: an experience with multiple inheritance. In *OOPSLA'91*, 1991.
- [Hal84] D. C. Halbert. Programming by example. Technical report, Xerox Office Systems Division, 1984.
- [Hen86] D.A. Henderson, Jr. The trillium user interface design environment. In *CHI'86*, 1986.
- [HH87] D. Hix and H.R. Hartson. A structural model for hierarchically describing human-computer dialogue. In *INTERACT'87, Second IFIP on Human-Computer Interaction*, 1987.
- [HH88] T.R. Henry and S.E. Hudson. Using active data in a uims. In *ACM UIST'88*, 1988.
- [HH89] H.R. Hartson and D. Hix. Human-computer interface development: Concepts and systems for its management. *Computing Surveys*, 21(1), 1989.
- [Hil86] R. D. Hill. Supporting concurrency, communication and synchronization in human-computer interaction the sassafras uims. *ACM Trans. on Graphics*, 5(4), 1986.
- [HK88] Scott E. Hudson and Roger King. Semantic feedback in the higgins uims. *IEEE Trans. on Soft. Eng.*, 1988.
- [HSL85] P.J. Hayes, P.A. Szekely, and R.A. Lerner. Design alternatives for user interface management systems based on experience with cousin. In *CHI'85*, 1985.



- [Jac83] R.J.K. Jacob. Using formal specification in design of human-computer interface. *CACM*, 26(4), 1983.
- [Jac86] R.J.K. Jacob. A specification language for direct-manipulation user interfaces. *ACM Trans. Graph.*, 5(4), 1986.
- [Kas82] D.J. Kasik. A user interface management system. *Comput. Graph.*, 16(3), 1982.
- [Kos89] A. Kobsa and W. Wahlster (eds). *User Models in Dialog System*. Springer-Verlag, 1989.
- [KP88] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. of Object-Oriented Programming*, 1(3):26–49, 1988.
- [Lee90] Ed Lee. User-interface development tools. *IEEE Software*, 1990.
- [LLM<sup>+</sup>92] G. Lobe, P. Lu, S. Melax, I. Parsons, J. Schaeffer, C. Smith, and D. Szafron. The enterprise model for developing distributed applications. Technical report, Department of Computing Science, University of Alberta, 1992.
- [LVC89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with interviews. *IEEE Computer*, 22(2), 1989.
- [Mac86] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans on Graphics*, 5(2), 1986.
- [Mac91] Wendy E. Mackay. Triggers and barriers to customizing software. In *CHI'91*, pages 153–160, 1991.
- [MCR91] Jock Mackinlay, Stuart K. Card, and George G. Robertson. A semantic analysis of the design space of input devices. *Human-Computer Interaction*, 5(2-3), 1991.
- [Mic90] Sun Microsystems. *Open Windows Developer's Guide 1.1*. Addison-Wesley, Reading MA, 1990.
- [Mor81] T. Moran. The command language grammar: A representation for the user interface of interactive systems. *Int. J. of Man-Machine Studies*, 15, 1981.

- [MW89] David L. Mausby and Ian H. Witten. Inducing programs in a direct manipulation environment. In *CHI'89*, 1989.
- [MWK89] David L. Mausby, Ian H. Witten, and K. A. Kittlitz. Metamouse: Specifying graphical procedures by example. In *SIGGRAPH'89*, 1989.
- [Mye88] Brad A. Myers. *Creating User Interface By Demonstration*. Academic Press, Inc., 1988.
- [Mye89] B.A. Myers. Encapsulating interactive behaviors. In *SIGCHI89*, pages 319–324, 1989.
- [Mye90] Brad A. Myers, et al. Garnet: Comprehensive support for graphical highly interactive user interfaces. *Computer*, 23(11):71–85, Nov. 1990.
- [NMK91] H. Nakatsuyama, M. Murata, and K. Kusumoto. A new framework for separating user interfaces from application programs. *SIGCHI Bulletin*, 23(1), 1991.
- [OD83] D.R. Olsen and E.P. Dempsey. Syngraph: A graphical user interface generator. In *SIGGRAPH'83 Proceedings*, 1983.
- [Ols84] D.R. Olsen. Pushdown automata for user interface management. *ACM Trans. Graph.*, 3(3), 1984.
- [Rat87] Martin Rathke. Dialogue issues for interactive recovery – an object oriented framework. In *Interact'87*, pages 745–750, 1987.
- [RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
- [Rei84] Steven P. Reiss. Graphical program development with pecan program development system. *ACM SIGPLAN Notices*, 19(5):30–42, 1984.
- [Rob83] Wilensky Robert. *Planning and Understanding - A Computational Approach to Human Reasoning*. Addison-Wesley, 1983.
- [Rub82] A. Rubel. Graphic based applications - tools to fill the software gap. *Digit. Des.*, 3, July 1982.

- [SBK85] J. Sibert, R. Belliardi, and A. Kamran. Some thoughts on the interface between user interface management systems and application software. In G.E. Pfaff, editor, *User Interface Management Systems*, Springer-Verlag, 1985.
- [Sch87] Kurt J. Schmucker. Macapp: An application framework. In R. M. Baecker and W. A. S. Buxton, editors, *Readings in Human-Computer Interactions*, pages 591-594, 1987.
- [SDB86] R.G. Smith, R. Dinitz, and P. Barth. Impulse-86: A substrate for object-oriented interface design. In *ACM OOPSLA '86*, 1986.
- [SG91] Gurminder Singh and Mark Green. Automating the lexical and syntactic design of graphical user interfaces: The uofa\* uims. *ACM Trans on Graphics*, 10(3):213-254, July 1991.
- [Sha90] Yen-Ping Shan. Mode: A uims for smalltalk. In *ECOOP/OOPSLA '90 Proceedings*, Oct. 1990.
- [SHB86] J.L. Sibert, W.D. Hurley, and T.W. Bleser. An object-oriented user interface management system. *Computer Graphics*, 20(3):259-267, 1986.
- [Sin89] Gurminder Singh. *Automating the Lexical and Syntactic Design of Graphical User Interfaces*. PhD thesis, University of Alberta, 1989.
- [SLN92] Pedro Szekely, Ping Luo, and Robert Neches. Facilitating the exploration of interface design alternative: The humanoid model of interface design. In *ACM CHI'92*, May 1992.
- [SRH85] A. J. Schulert, G. T. Rogers, and J. A. Hamilton. Adm - a dialogue manager. In *ACM CHI'85*, 1985.
- [SY88] M.L. Scott and S-K Yap. A grammar-based approach to the automatic generation of user-interface dialogues. In *ACM CHI'88 Proceedings*, 1988.
- [Sze88] Pedro Szekely et al. A user interface toolkit based on graphical objects and constraints. In *OOPSLA '88 Proceedings*, 1988.
- [Sze89] Pedro Szekely. Standardizing the interface between applications and uims's. In *ACM UIST'89*, Nov. 1989.

- [Sze90] Pedro Szekely. Template-based mapping of application data to interactive displays. In *ACM UIST'90*, Oct. 1990.
- [Tei75] W. Teitelman. *Interlisp Reference Manual*. Xerox PARC, Palo Alto, Calif., 1975.
- [Vit84] Jeffery S. Vitter. US&R: A new framework for redoing. *ACM SIGPLAN Notices*, 19(5):169-176, 1984.
- [Vli90] John M. Vlissides. *Generalized Graphical Object Editing*. PhD thesis, Stanford University, June 1990.
- [VPH83] J. VAN DEN Bos, M.J. Plasmeijer, and P.H. Hartel. Input-output tools: A language facility for interactive and real-time systems. *IEEE Trans. Softw. Eng.*, SE-9(3), 1983.
- [VT91] John M. Vlissides and Steven Tang. A unidraw-based user interface builder. In *ACM UIST'91*, 1991.
- [Was85] A. I. Wasserman. Extending transition diagrams for the specification of human-computer interaction. *IEEE Trans. Softw. Eng.*, SE-11(8), 1985.
- [Web89] Bruce F. Webster. *The NeXT Book*. Addison-Wesley, Reading, 1989.
- [WG91] Haiying Wang and Mark Green. An event-object recovery model for object-oriented interfaces. In *ACM UIST'91*, Nov., 1991.
- [WG93] Haiying Wang and Mark Green. A object-oriented framework for user customization. In *to appear on HCI'93 at Florida*, August, 1993.
- [WR82] P.C.S. Wong and E.R. Reid. Flair - user interface dialog design tool. *Computer Graphics*, 16(3), 1982.
- [Yan88] Yiya Yang. A new conceptual model for interactive user recovery and command reuse facility. In *ACM CHI'88*, pages 165-170, 1988.
- [Ye92] Chun Ye. Experiments in selective search extensions. Master's thesis, University of Alberta, 1992.