

# Design Patterns and Frameworks for Developing WIMP<sup>+</sup> User Interfaces

Vom Fachbereich Informatik  
der Technischen Universität Darmstadt  
genehmigte

## Dissertation

zur Erlangung des akademischen Grades der  
Doktor-Ingenieurin (Dr.-Ing.)

vorgelegt von

**Yongmei Wu, M.Sc.**

geboren in Guizhou, China

Referenten der Arbeit:

Prof. Dr.-Ing. Hans-Jürgen Hoffmann

Prof. Robert J.K. Jacob, Ph.D.

Tag des Einreichens: 09.10.2001

Tag der mündlichen Prüfung: 11.12.2001

D17

Darmstädter Dissertation 2001

## Abstract

This work investigates the models and tools for support of developing a kind of future user interfaces, which are partially built upon the *WIMP* (Windows, Icons, Menus, and Pointing device: the mouse) interaction techniques and devices; and able to observe and leverage at least one *controlled process* under the supervision of their user(s). In this thesis, they are called *WIMP<sup>+</sup> user interfaces*. There are a large variety of applications dealt with *WIMP<sup>+</sup> user interfaces*, e.g., robot control, telecommunication, car driver assistant systems, distributed multi-user database systems, automation rail systems, etc.

At first, it studies the evolution of user interfaces, deduces the innovative functions of future user interfaces, and defines *WIMP<sup>+</sup> user interfaces*. Then, it investigates *high level models* for user interface realization. Since the most promising user-centered design methodology is a new emerging model, it is still short of modeling methodology and rules to support the concrete development process. Therefore, in this work, a universal modeling methodology, which picks up the design pattern application, is researched and used to structure different *low level user interface models*. And a framework, named *Hot-UCDP*, for aiding the development process, is proposed. Among the design patterns required by *Hot-UCDP*, this work puts its most effort on investigating user interface software architectural patterns. As a result, a *WIMP<sup>+</sup> user interface* software architectural pattern *Acquisition-Computation-Expression-Execution* (*ACEE*) was discovered. Tools for user interface implementation are also surveyed. Based on the research results, an *ACEE*-based software framework prototype called *Hot-WIMP<sup>+</sup>* was developed. To support programming *WIMP<sup>+</sup> user interfaces* from multiple abstract levels, *Hot-WIMP<sup>+</sup>* provides not only white and black box technique but also visual tools, e.g., *Acquisition Definer*, *Expression Specification Tools*, *Execution Definer*.

The practicability of *Hot-UCDP*, *ACEE*, and *Hot-WIMP<sup>+</sup>* is proven by applying them to create three *WIMP<sup>+</sup> user interfaces*, *LLDemo*, *Hot-WebRobi*, and *Hot-Demo* for controlling a model robot. During their development, a robot application domain pattern *SelectMe-ConveyMe-SettleMe* and a human and computer interaction pattern *Two-handed Manipulation* were well researched and documented. *SelectMe-ConveyMe-SettleMe* could be, in principle, used as a starting point to develop other *WIMP<sup>+</sup> user interfaces* for controlling the end-effector of a 3-D robot. *Two-handed Manipulation* could also be used to guide the design of bimanual interaction. The effect and efficiency of these *WIMP<sup>+</sup> user interfaces* for support of users fulfilling telecontrol tasks are also investigated through a case study. Additionally, the combination of the Internet and telerobotics is also considered within this work.

## Zusammenfassung (German)

Die vorliegende Arbeit beschäftigt sich mit den Modellen und Werkzeugen für die Entwicklung einer zukünftigen Benutzungsschnittstelle. Diese Benutzungsschnittstelle basiert teilweise auf den interaktiven Methoden und Geräten von WIMP (engl. Windows, Icons, Menus, and Pointing device: the mouse). Sie kann mindestens einen *kontrollierten Prozeß* unter Überwachung von Benutzern beobachten und regeln. Hier wird sie *WIMP<sup>+</sup>-Benutzungsschnittstelle* genannt. Die *WIMP<sup>+</sup>-Benutzungsschnittstelle* wird in zahlreichen Anwendungen, z.B. Roboterkontrolle, verteilte Multibenutzerdatenbanksysteme, Telekommunikation, Fahrerassistenzsysteme, automatische Bahnführungssysteme, usw., benutzt.

Aus der Auswertung der zahlreichen Quellen wurden zuerst die innovativen Funktionen der zukünftigen Benutzungsschnittstellen hergeleitet und die *WIMP<sup>+</sup>-Benutzungsschnittstelle* definiert. Dann wurden die *High-Level-Modelle* für Realisierung der Benutzungsschnittstellen untersucht. Da sich die vielversprechende benutzerzentrierte Entwurfsmethodik (engl. user-centered design methodology) noch in der Entwicklungsphase befindet, fehlen noch Modellierungsmethoden und Vorschriften für die Unterstützung des konkreten Entwicklungsprozesses. Deshalb wurde eine universale Modellierungsmethodik, die das Verwenden von Entwurfsmustern aufgreift, in dieser Arbeit erforscht und zum Strukturieren der verschiedenen *Low-Level-Modelle* der Benutzungsschnittstellen eingesetzt. Zur Unterstützung des Entwicklungsprozesses wurde ein Modell *Hot-UCDP* erstellt. In den von *Hot-UCDP* erforderten Entwurfsmustern legte diese Arbeit großen Wert auf die Untersuchung der Softwarearchitekturmuster der Benutzungsschnittstellen. Als ein Ergebnis wurde ein Softwarearchitekturmuster *ACEE* (engl. Acquisition-Computation-Expression-Execution) für *WIMP<sup>+</sup>-Benutzungsschnittstellen* entwickelt. Werkzeuge für die Implementation der Benutzungsschnittstellen wurden auch recherchiert. Basierend darauf wurde ein Softwareprototyp des Anwendungsrahmens *Hot-WIMP<sup>+</sup>*, der auf *ACEE* beruht, entwickelt. Um das Programmieren der *WIMP<sup>+</sup>-Benutzungsschnittstellen* von verschiedenen abstrakten Ebenen aus zu unterstützen, liefert *Hot-WIMP<sup>+</sup>* nicht nur Verfahren von White- und Black-Box-Methoden sondern auch visuelle Werkzeuge, z.B. *Acquisition-Definer*, *Expression Specification Tools*, *Execution-Definer*.

Die Anwendbarkeit von *Hot-UCDP*, *ACEE*, und *Hot-WIMP<sup>+</sup>* wurde durch die Erstellung von drei *WIMP<sup>+</sup>-Benutzungsschnittstellen*, nämlich *LLDemo*, *Hot-WebRobi* und *Hot-Demo*, für die Kontrolle eines Modellroboters gezeigt. Bei deren Entwicklung wurde ein Muster im Anwendungsbereich des Roboters, *SelectMe-ConveyMe-SettleMe*, und ein Muster der Mensch-Maschine-Interaktion, *Two-handed Manipulation*, erforscht und dokumentiert. Das Muster *SelectMe-ConveyMe-SettleMe* kann im Prinzip als Ausgangspunkt für die Entwicklung der anderen *WIMP<sup>+</sup>-Benutzungsschnittstellen* für die Kontrolle des Endeffektors eines 3-D-Roboters eingesetzt werden. Das Muster *Two-handed Manipulation* kann auch beim Entwurf der bimanuellen Interaktion verwendet werden. Die Auswirkung und die Effizienz dieser *WIMP<sup>+</sup>-Benutzungsschnittstellen* bei der Unterstützung der Durchführung der Aufgaben der Fernkontrolle wurden ebenfalls untersucht. Die Kombination des Internets und der Telerobotik wurde in dieser Arbeit zusätzlich auch berücksichtigt.

## Foreword

This doctoral work was done during my research activity at the Chair Programming Languages and Compilers of the Darmstadt University of Technology.

It was Prof. Dr. Hans-Jürgen Hoffmann, head of the Programming Languages and Compilers group, who made it possible for me to execute this work. He let me realize how much better the term “*doctoral father*” describes the nurturing role of a supervisor. I am very grateful to his great support and successful guidance to this work.

I feel very much indebted to my second supervisor Prof. Dr. Robert J.K. Jacob from the Tufts University for assessment of this work and valuable feedback.

I especially want to thank my colleagues at the Darmstadt University of Technology. Special thanks are due to Dr. Elke Siemon for her good cooperation, help, and encouragement. Other colleagues, Jan Weerts, Patrick Closhen, Daniela Handl, Ludger Martin, Gerlinde Hess, and Marion Braun also gave me helps in many ways. I appreciate them.

I also want to thank the students at the Darmstadt University of Technology who did student research projects under my guidance which in turns made contributions to this work. They are Martin Friedmann, Erik Hansen, and Wolfgang Hess.

I appreciate deeply to the research support given by Prof. Dr. Oskar von Stryk, Dr. Ingrid Biehl, and Dr. Ulrike Brandt.

This work was supported by FAZIT-Stiftung, Promotions-Abschlussförderung of the Darmstadt University of Technology, and Frauenförderung of the Computer Science Department of the Darmstadt University of Technology. I thank them very much.

Last but not least, I would like to thank my parents Chengjun Wu and Fanying Kong for making me go on this way. Completing a doctoral dissertation means persistence, patience, and hard work, especially for a women with a small child newly coming to a foreign country. Without the understanding and effective support of my husband Chunyang Xie, who was also at the same time working at his doctoral thesis at the Automotive Engineering Department of the Darmstadt University of Technology, as well as my cute daughter Jing Xie, it would not have been possible for me to proceed this work during this time. I thank them deeply.

Yongmei Wu

Wolfsburg-Fallersleben, Germany, July 2001

## Contents

<b>1</b>	<b>INTRODUCTION.....</b>	<b>1</b>
1.1	User Interface: a Vita Factor to a Contemporary and Future Application.....	1
1.2	New Challenges in Contemporary User Interface Research.....	2
1.3	Goals of this Work .....	2
1.4	Organization of this Thesis .....	3
<b>2</b>	<b>USER INTERFACES AND WIMP<sup>+</sup> USER INTERFACES .....</b>	<b>6</b>
<b>2.1</b>	<b>Evolution of User Interfaces .....</b>	<b>6</b>
2.1.1	Traditional User Interfaces.....	6
2.1.2	Features and Resultant Deficiencies of Traditional User Interfaces .....	8
2.1.3	Non-WIMP User Interfaces .....	11
<b>2.2</b>	<b>WIMP<sup>+</sup> User Interfaces.....</b>	<b>14</b>
2.2.1	Supervisory Control.....	14
2.2.1.1	Supervisory Control in Automation and Control Systems .....	14
2.2.1.2	Supervisory Interactive Computer Systems .....	15
2.2.2	What is a WIMP <sup>+</sup> User Interface? .....	18
<b>2.3</b>	<b>Examples.....</b>	<b>19</b>
2.3.1	User Interfaces for Robot Control.....	19
2.3.2	User Interfaces of WAP Mobile Phones .....	20
2.3.3	User Interfaces of Car Driver Assistant Systems .....	22
<b>2.4</b>	<b>Characteristics of WIMP<sup>+</sup> User Interfaces.....</b>	<b>23</b>
<b>2.5</b>	<b>Summary .....</b>	<b>25</b>
<b>3</b>	<b>THE STATE-OF-THE-ART MODELS AND TOOLS FOR WIMP<sup>+</sup> USER INTERFACES .....</b>	<b>26</b>
<b>3.1</b>	<b>User Interface Models .....</b>	<b>26</b>
3.1.1	High Level Models .....	26
3.1.1.1	Life Cycle Model .....	27
3.1.1.2	Usability Engineering .....	28
3.1.1.3	User-Centered Design Methodology .....	28

3.1.2	Design Patterns: a Powerful Modeling Methodology.....	30
3.1.2.1	The Context of User Interface Modeling Methodology .....	30
3.1.2.2	The Root of Design Patterns .....	32
3.1.2.3	Design Patterns in Software Engineering.....	35
3.1.2.3.1	The Scale of Software Patterns .....	37
3.1.2.4	Design Patterns in HCI .....	38
3.1.2.5	Application Domain Patterns.....	41
3.1.3	Hot-UCDP: A Framework for User-Centered Design Methodology.....	43
3.1.4	Software Architectural Patterns for User Interface Development.....	45
3.1.4.1	Seeheim Model.....	45
3.1.4.2	Presentation-Abstraction-Control.....	46
3.1.4.3	Model-View-Controller .....	48
<b>3.2</b>	<b>Tools .....</b>	<b>49</b>
3.2.1	Window Management Systems .....	50
3.2.2	Toolkits.....	51
3.2.3	User Interface Design Systems.....	51
3.2.4	Software Frameworks.....	52
<b>3.3</b>	<b>Pattern-based User Interface Frameworks: an Effective and Efficient Way for User Interface Development .....</b>	<b>52</b>
<b>3.4</b>	<b>Implications for WIMP<sup>+</sup> User Interfaces.....</b>	<b>54</b>
<b>3.5</b>	<b>Summary .....</b>	<b>55</b>
<b>4</b>	<b>ACEE: A SOFTWARE ARCHITECTURAL PATTERN FOR DEVELOPING WIMP<sup>+</sup> USER INTERFACES .....</b>	<b>56</b>
<b>4.1</b>	<b>About the Pattern Description.....</b>	<b>56</b>
<b>4.2</b>	<b>The ACEE Pattern .....</b>	<b>57</b>
<b>4.3</b>	<b>Summary .....</b>	<b>76</b>
<b>5</b>	<b>HOT-WIMP<sup>+</sup>: AN ACEE-BASED SOFTWARE FRAMEWORK .....</b>	<b>78</b>
<b>5.1</b>	<b>Application Framework of VisualWorks Smalltalk .....</b>	<b>78</b>
<b>5.2</b>	<b>The Programming Layers of Hot-WIMP<sup>+</sup>.....</b>	<b>80</b>
<b>5.3</b>	<b>Components of Hot-WIMP<sup>+</sup> .....</b>	<b>80</b>
5.3.1	Fundamental Classes.....	81
5.3.1.1	Classes for Establishing Low Level Connection with Sensor and Effector .....	81
5.3.1.1.1	AcceptorConnectorAM .....	81



6.8.1	Evaluation on LLDemo, Hot-WebRobi, and Hot-Demo .....	136
6.8.2	Evaluation on Hot-UCDP, ACEE, and Hot-WIMP+ .....	138
<b>6.9</b>	<b>Other Applications.....</b>	<b>139</b>
<b>6.10</b>	<b>Summary .....</b>	<b>139</b>
<b>7</b>	<b>CONCLUSIONS AND FUTURE WORK .....</b>	<b>141</b>
<b>7.1</b>	<b>Conclusions.....</b>	<b>141</b>
<b>7.2</b>	<b>Future work.....</b>	<b>145</b>
	<b>Appendix A: Glossary .....</b>	<b>147</b>
	<b>Appendix B: Typographic Convention.....</b>	<b>153</b>
	<b>Appendix C: The Robot Client and the Robot Server Communication Languages .....</b>	<b>154</b>
	<b>Appendix D: The List of Figures.....</b>	<b>159</b>
	<b>Appendix E: The List of Tables .....</b>	<b>161</b>
	<b>REFERENCES .....</b>	<b>162</b>



# 1 Introduction

## 1.1 User Interface: a Vita Factor to a Contemporary and Future Application

With computer hardware cost rapidly falling down, presently significant computing capability is available almost everywhere. Not only office work but also automation and control, robotics, telecommunication, e-commerce, and so on, can not be well handled without the help of computers. Computers have been becoming a kind of basic tools for human life.

Indeed, more and more people world wide with different culture, religion, knowledge, gender, age have being involved in use of computers (e.g., [63] [79]). Therefore, user interfaces, which are used to support a very large amount of users interacting with computers, are extremely critical to contemporary and future applications due to the following three major reasons:

- A user interface determines if users like or dislike an application and how much they may follow their skill and ingenuity with it, which in turns decides the destiny of an application (e.g., [59]).
- A user interface decides the safety of a life critical application. For example, in an application of nuclear power plant automation, airplane control, or surgical operation, a poor user interface may cause catastrophe (e.g., [67] [96]).
- A user interface determines the cost of an application since on average about 50% of the code and implementation time of an application is devoted to its user interface part and more than two-fifths of the software maintenance is due to the poor user interface (e.g., [71] [74]).

## 1.2 New Challenges in Contemporary User Interface Research

Graphical User Interfaces (*GUIs*) are the dominant vehicle for modern Human and Computer Interaction (*HCI*). Since within GUIs users depend on the Windows, Icons, Menus, and Pointing device: the mouse (*WIMP*) interaction techniques and devices to interact with computers, GUIs are also dubbed *WIMP user interfaces*. Although WIMP user interfaces occupy the reputation of intuitivity, ease to learn and use, they are office work oriented. They are not optimal to support human interacting with computer especially as the application domains go beyond the desktop metaphor [37] [80] [89] [96], e.g., in automation and control, in robotics, in telecommunication, in aviation, etc. They have neither appropriately encompassed the new emerging interaction devices (e.g., eye trackers, head-coupled displays, and audio interaction devices) and techniques (e.g., sound interaction, two-handed manipulation, gesture, animation, and ubiquitous computing) to support HCI, nor embedded new research results of human factors.

Therefore, research work on future generation user interfaces has been strongly appealed since early 1990s [29] [40] [80].

To create future user interfaces, user interface researchers are confronted with large amount of challenges dealing with multiple disciplines such as computer science, psychology, sociology, anthropology, and industrial design [1]. They are now investigating, e.g.,

- The form of the future user interfaces, i.e., “*like what the future user interfaces look?*” (e.g., [29] [37] [80]).
- The new interaction techniques and devices for future HCI (e.g., [29] [100] [104]).
- The social effects and psychology affects on their users (e.g., [81] [96]).
- The software models and tools for support of their constructions (e.g., [56]).

## 1.3 Goals of this Work

This doctoral work researches future user interfaces from the user interface engineering point of view. That is, it investigates models and tools to support developing a kind of future user

interfaces. Although there are a known software model and tool for creating a kind of future user interfaces [56], they are specific to support the “*finer-grained*” programming level. Creating future user interfaces needs the support of models and tools at different abstract levels: they should be able to effectively support not only the implementation phase but also the whole development process. Accordingly, this work investigates the related models and tools from an overall point of view.

The goals of this work can be unfolded as follows:

- *Study the evolution of user interfaces. Research future user interfaces and deduce their possible innovative functions. Elaborate on and define a kind of future user interfaces.*
- *Study the state-of-the-art models and tools for support of developing user interfaces, especially for the new defined user interfaces. The models and tools should strive to cover different abstract levels for the development.*
- *Study the user interface modeling methodology. Research the universal modeling methodology: design patterns, and its applications. Adopt it to help establish different user interface models.*
- *Research and develop a software architecture for building the new defined user interfaces. Document it as a design pattern.*
- *Develop a software framework based on the new discovered pattern. The framework should strive to provide multi layer programming technique for reuse.*
- *Demonstrate and investigate the utility of the design patterns and frameworks developed by this work through development practice.*

## **1.4 Organization of this Thesis**

The reminding contents of this thesis is arranged as follows:

- Chapter 2 will briefly survey the evolution of user interfaces and reason about why research of future user interfaces is necessary. It will deduce the major innovative

functions and characteristics of future user interfaces and define what is a *WIMP<sup>+</sup> user interface*. Important features of *WIMP<sup>+</sup> user interface* will also be studied through three examples.

- Chapter 3 will overview the state-of-the-art models and tools for developing *WIMP<sup>+</sup> user interfaces*. It will present the models in support of both the whole development process and phases of user interface engineering. Especially, the attractive universal modeling methodology, design patterns, and its state-of-the-art application in software engineering, HCI design, and application domain will be surveyed. A framework, named *Hot User-Centered Design Pattern System (Hot-UCDP)* proposed by this work for support of user interface engineering, will be introduced. The state-of-the-art software architectural patterns for user interface development will be studied. Different tools, e.g., window manager system, toolkits, user interface design system, and software framework, will also be overviewed there.
- Chapter 4 will introduce a quite mature software architectural pattern, *Acquisition-Computation-Expression-Execution (ACEE)*, for support of creating *WIMP<sup>+</sup> user interfaces*. It is documented by the thesis author and was presented at PLoP1999 [128]. In order to provide a sound software architecture for *WIMP<sup>+</sup> user interfaces*, a renewed version of *ACEE*, which is revised according to the comments and suggestions for improvement obtained from the writer's workshop at PLoP1999 and other resources, will be presented there.
- Chapter 5 will elaborate on *Hot-WIMP<sup>+</sup>*, an *ACEE*-based object-oriented software framework prototype designed and implemented by the thesis author herself during this work. It is a software framework for aiding in programming *WIMP<sup>+</sup> user interfaces*. Its multi layer programming technique will be introduced there in detail.
- Chapter 6 will demonstrate the utility of *Hot-UCDP*, *ACEE*, and *Hot-WIMP<sup>+</sup>* in the development of *WIMP<sup>+</sup> user interfaces* through a robot control application. Accordingly, a robot application domain pattern *SelectMe-ConveyMe-SettleMe* and a HCI pattern *Two-handed Manipulation* which are revised from a workshoped paper [133] of the thesis author will be introduced. Three *WIMP<sup>+</sup> user interfaces*, *LLDemo*, *Hot-WebRobi*, and *Hot-Demo* developed by use of *Hot-*

*UCDP, ACEE, Hot-WIMP<sup>+</sup>* as well as *SelectMe-ConveyMe-SettleMe* and *Two-handed Manipulation* will be presented there. To study the effect and efficiency of the technique that these *WIMP<sup>+</sup> user interfaces* provide for aiding the end users to fulfill their task, a case study will be elaborated there, too.

- Finally, conclusions and future work will be discussed in chapter 7.

After that will be the appendices and references of this thesis. Appendix A is an explanation of the glossary used in this thesis. Appendix B interprets the typographic conversion of this thesis. Appendix C presents *the robot client and the robot server communication languages* used by those *WIMP<sup>+</sup> user interfaces* of the robot control example introduced in chapter 6. Appendix D will list all the figures used by this thesis. List of the tables will be presented in Appendix E.

## **2 User Interfaces and WIMP<sup>+</sup> User Interfaces**

### **2.1 Evolution of User Interfaces**

#### **2.1.1 Traditional User Interfaces**

User interfaces are that portions of interactive computer systems that communicate with the users [58]. From users' point of view, a user interface is composed of three basic elements: interaction devices, interaction tasks, and interaction techniques [35]. Interaction devices are the peripheral hardware of a computer system for HCI. They can be further divided into input devices and output devices. Interaction tasks are meaningful steps of users' task, e.g., selecting is an interaction task of the task delete a file. Interaction techniques are ways to use interaction devices to perform interaction tasks. To perform the same interaction task, there may be several interaction techniques with the same interaction devices. For example, in Windows NT 4.0, to delete file1, one can use the mouse to select the icon of file1, drag and drop it to the trash; or use the mouse to select the file1 icon at first, then issues the delete command integrated within the operation menu of file1.

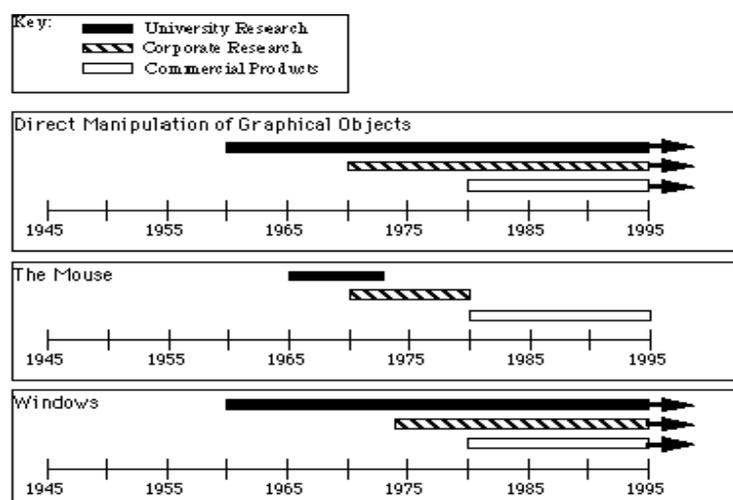
According to interaction devices, interaction tasks, and interaction techniques, traditional user interfaces can be divided into three generations [29] [80].

In the first generation, applications can only be run in the batch mode. Generally, users can not interact directly with computers except some privileged specialists load punched cards or pick up results from line printer output before or after the running of applications. For example, at the beginning, in ENIAC I, applications can only be loaded via a IBM card reader and a card punch is used for output of the results [120]. Consequently, interaction tasks were not clearly

defined in the first generation. Interaction devices, if exist, are the mechanical switches, jumper wires, and lights.

Second generation user interfaces are keyboard oriented, alphanumeric typewriters and (later) full screen terminals. They are also known as the command line user interfaces [29] [80]. By use of a keyboard as input device and an alphanumeric display as output, a user can at the first time interact with a computer via commands (e.g., MS-DOS, Unix). To accomplish an interaction task, a user must input a command or a command sequence rigidly. For example, to delete file1 in MS-DOS, one must input the command `del file1`. And if the system is busy on executing another time consuming task, the user has to wait until the finish of the task. Furthermore, to communicate with second generation user interfaces, users have to remember the complex commands. Therefore, except professional people, most users do not like second generation user interfaces because they are difficult to learn and use as described above.

The investigation of direct-manipulation (first demonstrated by Ivan E. Sutherland in his Ph.D. thesis Sketchpad in 1963 at MIT [103]), the invention of the Mouse (the first mouse prototype was invented in 1964 by Douglas Engelbart at the Stanford Research Laboratory [48]), the exploration of Windows (in 1968 multiple tiled windows were demonstrated by Douglas Engelbart; in 1969 at the University of Utah, Alan Kay in his Ph.D. thesis proposed the overlapping windows), the emerging of raster display, etc., make the development of third generation user interfaces possible.



**Fig. 1: The development and use of the main WIMP user interface techniques and devices in academia and industry [75]**

Third generation user interfaces are the traditional graphical user interfaces which, as described in section 1.2, are also called WIMP user interfaces. Fig. 1 shows the research and application of the main WIMP interaction techniques and devices in academia and industry. It is because the mature of these interaction techniques and devices, the first industrial prototype of WIMP user interfaces was successfully developed on a broad basis by Xerox PARC in the middle of 1970s [37]. Afterwards, the WIMP family, Macintosh, Microsoft Windows, IBM OS/Warp, etc., came into the world. And they have become the dominant vehicle for present HCI.

Unfortunately, although WIMP user interfaces are easier to learn and use compared with command line user interfaces, although they have been widely accepted and used in office work, they are not optimal [29] [37] [80]. Users still keep complaining about the failure of their design (e.g., [89]).

### **2.1.2 Features and Resultant Deficiencies of Traditional User Interfaces**

From first generation user interfaces to present WIMP user interfaces, user interface development has been experienced almost 50 years. Each generation was built under the constraints of hardware and experience of people of a time period. And each successor is more “*user friendly*” than its precursor. Obviously, WIMP user interfaces are the outstanding representatives of traditional user interfaces. Therefore, only the features and resultant deficiencies of traditional WIMP user interfaces are elaborated here:

- *Syntactic command based*

WIMP user interfaces employ only two kinds of syntactic commands for HCI [80]. One inherits that of command line user interfaces as MS-DOS. The other uses direct-manipulation of windows, icons, menus with the mouse. Although this direct-manipulation technology provides users with more intuitive interaction style, it is still a kind of commands. For example, to delete a file in a WIMP user interface, a user still issues a syntactic command. Only the command is expressed in a more intuitive way by use of icon and operation menu. Although this direct-manipulation technology is easier to learn and use than the interaction technology used by command line user interfaces, in

some cases, especially for the experienced users, it is less convenient<sup>1</sup>. Shortly, both kinds of commands are not optimal to users.

- *Limited interaction devices and techniques*

Keyboard, the mouse, and a 2-D display are the standard input and output devices of WIMP user interfaces. With keyboard and the mouse as input, only a few simple discrete events for HCI can be generated in a second to a computer. This low bandwidth interaction channel has become a barrier of modern high speed computer systems. On the other hand, keyboard and the mouse are not optimal for users interacting with computers since they limit some potential interaction abilities of users. For example, it allows only the dominant hand of a user to do manipulation, while people are used to work with two hands. Even worse, they cause stress to users. According to an orthopedist, many people who work with computers suffer from shoulder aches. And the left shoulder is worse than the right among 80 percent of them. No matter how accurate this statement is, it implies that *unbalanced workload to two hands is an injury to users*. The interaction style of keyboard, the mouse, and 2-D display is not ideal to normal users, and even worse to physical or other disabled people.

- *Only text and 2-D support*

Text and 2-D of WIMP user interfaces serve their major goal of supporting office work pretty well since in most cases applications of office work need only text and 2-D graphics support, e.g., document layout, word processing, spreadsheets, etc. But when applications go beyond the office work, text and 2-D show some pitfalls since people live in a 3-D world and are used to 3-D interaction. Mapping 3-D information to a 2-D flat display is difficult and indirect for both users and designers.

- *Pure users' control*

---

<sup>1</sup> For example, to find a file, a user has probably to spend lots of time to navigate the user interface in order to find the file.

WIMP user interfaces require users to be in control at all the time [37]. Every interaction task must be entered by a user. If the user stops input, then interaction stops. This interaction style is neither natural nor convenient. The more complex the task, the harder and more tedious are interactions for a user. Moreover, it requires that a user has to be conscious at both tasks and dialogue possibilities at the same time. Since to issue a command, a user has to observe actively the interaction possibilities provided in the user interfaces first, then may do manipulation. It violates the rule of experimental psychology “*people can pay attention only one thing at any moment* [89].” *Good user interfaces should be able to observe the users and/or the task environments, leverage<sup>2</sup> the task environments, and deduce the optimal interaction possibilities to users.*

- ... (There are more features and resultant deficiencies coming, the history is going on.)

More than 20 years have passed since Xerox PARC invented the first WIMP user interface. The limitations of hardware and experience of that time have been broken down. For example,

- Computers have become commodities. Even household computers are more powerful and faster than the research computers at the time when the first WIMP user interface was invented.
- Richer and higher bandwidth interaction devices like eye trackers, head mounted displays, tactile sensors, multiple degrees of freedom data gloves, etc. provide users with greater possibility to interact with computers than keyboard and the mouse.
- The Internet, Intranet, and wireless network link most computers together. Wireless Application Protocol (*WAP*) [119], a de facto specification for providing the Internet communication and advanced telephony service on mobile phones, pagers, laptops, and other wireless terminals will bring new possibilities for nomadic computing to users.
- More and more application domains call for use of computers, e.g., e-commerce (e.g., [49]), telecommunication (e.g., [125]), robotics (e.g., [13]), automation and control

---

<sup>2</sup> Leverage means control or modify.

(e.g., [97]), WAP and WWW applications (e.g., [72]), car driver assistant systems (e.g., [122]), etc.

Now, enough momentum for developing new generation user interfaces has been accumulated. As in the middle of the 1970s, an era of new generation user interfaces is coming.

### 2.1.3 Non-WIMP User Interfaces

In early 90s, the idea to research future interfaces has already dawned. In August 1990, at the workshop of Software Architectures and Metaphors for Non-WIMP user interfaces of ACM SIGGRAPH'90 [40], several user interface researchers from academia and industry have proposed and discussed the research work on new generation user interfaces. They call future user interfaces *Non-WIMP user interfaces*. The term *Non-WIMP* refers to any interface style that is *not* based on the desktop metaphor. Afterwards, several ideas and research works on Non-WIMP user interfaces have emerged respectively, e.g., “*Non-command user interface*” [80], “*Post-WIMP user interfaces*” [29], etc.

In [29] [37] [40] [80] [83] several user interface researchers have elaborated on their visions of Non-WIMP user interfaces from diverse aspects. In summary, the interaction tasks, techniques, and devices of Non-WIMP user interfaces will distinguish greatly from that of traditional user interfaces, as depicted in Table 1.

No doubt, the available technologies and new research results of HCI will be used to build Non-WIMP user interfaces to overcome the deficiencies of their precursors. However, it does not mean that Non-WIMP user interfaces are the locus for simply accumulating new technologies. “*Instead of technique-oriented or device-oriented, they should be user and task oriented*” [80]. *They should provide clearly “Sites, Modes, and Trails” [82] to support their users interacting with computers.* That is, adequate interaction techniques and devices should be tailored to a Non-WIMP user interface according to the community of users and the tasks of application domains.

**Table 1: Interaction tasks, techniques, and devices of user interfaces**

	Interaction Tasks	Interaction Techniques	Interaction Devices
The First Generation User Interfaces	None	Manually interrupting the running of an application	Mechanical switches, jumper wires, lights
Command Line User Interfaces	Functional commands, numbers, text	Character command based on keyboard input	Keyboard, alphanumeric full screen monitor
WIMP User Interfaces	Two dimensional discrete visual commands and interaction tasks in Command Line User Interfaces	Direct-manipulation via windows, icons, menus, and pointing device: the mouse and interaction techniques in Command Line User Interfaces	Keyboard, the mouse, CRT monitor
Non-WIMP User Interfaces	Explicit interaction tasks as in their precursors and implicit ones such as rotating a 3-D object	All possible interaction techniques suitable for their interaction tasks, e.g., voice and gesture interactions	All possible digital devices including new and novel ones, e.g., pen, eye tracker, head-mounted display, data glove, 3-D sensor

Based on the above hypothesis, Non-WIMP user interfaces should provide at least the following innovative functions to users:

- *More powerful and natural languages for users interacting with computers*

In addition to syntactic commands, Non-WIMP user interfaces will take advantage of other communication abilities of people that has not been used in their precursors before, e.g., speech, hearing, and gesture. They should include but not limit to: 1) natural language, the most natural and powerful communication vehicle of users; 2) visual task-oriented programming languages, such as programming by demonstration, imperative graphical language, controlflow, dataflow, workflow, etc., usually specific to application

domains and often ideal to novice users; 3) scripting languages, (as, e.g., Tcl/Tk [88]), a promising tool for experienced users.

- *Richer interaction devices and techniques*

Non-WIMP user interfaces will use new interaction devices and techniques like eye trackers, pens, tactile sensors, sound, and gesture for input and head mounted displays, wall size displays, sound, 3-D virtual space for output. They can also process diverse, possibly parallel and real-time, interaction information arrived from any corner of the world via the Internet or other communication channels. Therefore, in addition to dominant hand, users could use their other organs to interact with computers in a more flexible way. For example, eye gaze interaction, gesture interaction, haptic interaction, two-handed manipulation, etc.

- *Agents*

Non-WIMP user interfaces will equip users with diverse knowledge based agents for alleviating their interaction workload. After acquiring the intentions of users, agents can release users from most tedious interaction tasks. Some agents can even entirely replace users to do some dangerous and harmful work.

- *Supervisory control*

Unlike the pure users' control of their precursors, Non-WIMP user interfaces should be able to automatically observe their users and the task environments, understand the analog input from diverse sensors such as video, sound, and gesture ones that attach to them, leverage the task environments and deduce the optimal dialogue possibility to users according to the integrated knowledge bases specific to application domains. That is, they can possibly automatically leverage the task environments to fulfil the users' tasks under the supervision of their users.

To build up a mature Non-WIMP user interface with all these functions, no doubt, needs comprehensive, long time, and hard research work of all user interface researchers.

*Instead of investigating all the facilities of agents, natural language interaction, 3-D visualization, or other novel functions of Non-WIMP user interfaces, this thesis concentrates upon one topic and will investigate those user interfaces for supervisory control.*

## **2.2 WIMP<sup>+</sup> User Interfaces**

### **2.2.1 Supervisory Control**

#### **2.2.1.1 Supervisory Control in Automation and Control Systems**

The term supervisory control comes from automation and control systems [97]. Generally, present automation and control systems are composed of users (control engineers, operators), computers, controlled processes (task environments), sensors, and effectors. Sensors are responsible for capturing data from the controlled processes while leveraging the processes are the obligations of effectors. Computers, which initiate effectors and deduce the control decisions according to data coming from the sensors, integrated knowledge bases of the controlled processes, and users' instructions, are the kernel of the systems. Users, depending on the system design, may heavily or may not be coupled with controlled processes. Real-time response and feedback are critical aspects to the systems.

*“In the strictest sense, supervisory control means that one or more human operators are intermittently programming and continually receiving information from a computer that itself closes an autonomous control loop through artificial effectors and sensors to the controlled process or task environment.” [97, p2]*

*“In a less strict sense, supervisory control means that one or more human operators are continually programming and receiving information from a computer that interconnects through artificial effectors and sensors to the controlled process or task environment.” [97, p2]*

In automation and control systems, “*supervisory control*” implies users and they are possible to take the following responsibilities [97]:

- Planning what tasks should be done and how to do them better.
- Programming the computer to fulfil the tasks.
- Monitoring the system to make sure that all are working as planned and to detect failure.
- Intervening the control if the desired goal has been reached satisfactorily, or taking over the control in emergencies to specify a new goal or reprogram a new procedure.

### 2.2.1.2 Supervisory Interactive Computer Systems

Usually, in automation and control systems, a controlled process as described in above section touches with these artifacts:

- Controlled objects such as robot, tank, industrial boiler, chemical reaction container, and vehicle;
- Sensors such as temperature sensor, sound sensor, tactile sensor, and 3-D sensor;
- Effectors such as motor, switch, and actuator.

The statuses of the controlled objects are dynamic and influenced by some external factors, e.g., temperature, humidity, force, velocity, vision, voice, etc.; and can be detected by sensors and leveraged by effectors.

Moreover, in other computer systems out of automation and control, there are many entities touch the similar artifacts and possess the identical features as that of the controlled process described above, e.g., a distributed database, a WWW server, a multi-user documentation, and so on. These entities deal with also *controlled objects* whose current statuses can be detected by *sensors*<sup>3</sup> and leveraged by *effectors*<sup>4</sup>. For example, in a distributed database

---

<sup>3</sup> A sensor, here, is not limited to a hardware. It could be a programming interface of a software component as well.

<sup>4</sup> As a sensor, an effector could be a hardware or a programming interface of a software component.

system, several users are manipulating a database simultaneously via computer terminals. For each user, there is a mechanism acting as a *sensor* to detect the change made by others and update the display in time. And at the same time, a mechanism acting as an *effector* is provided to modify the database according to the change made by each user. Obviously, the database has already dealt with similar artifacts and possessed the identical features as that of the controlled process.

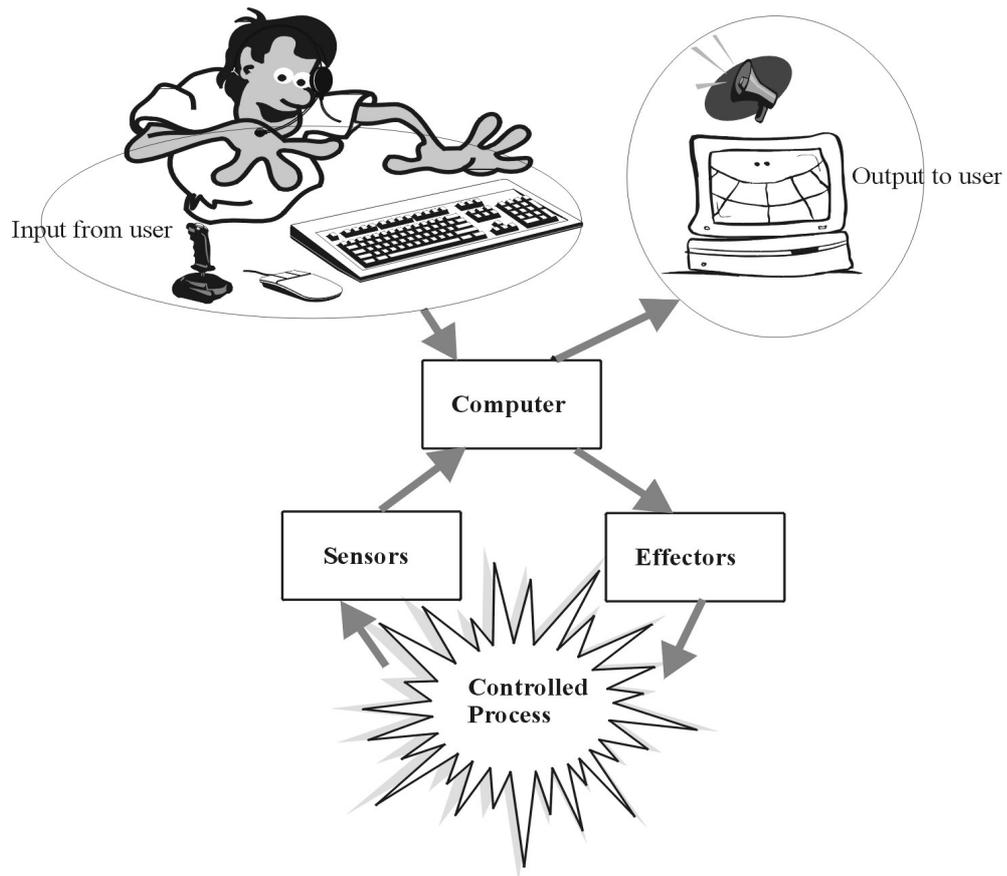
*In this thesis, any entity is defined as a controlled process if and only if it possesses dynamic status(es) that can be detected by sensors and leveraged by effectors.*

Hence, in terms of this definition, tanks, industrial boilers, distributed databases, WWW servers, etc. can be looked as *controlled processes*.

*If an interactive computer system is connected with at least one controlled process via both sensor(s) and effector(s) which are used to observe and/or leverage the process, and one obligation of its user(s) is to supervise if the task is fulfilled as required, then, in this thesis, the computer system is called as a supervisory interactive computer system.*

According to the definition, a supervisory interactive computer system can be depicted as Fig. 2. It consists of user(s), a computer, *controlled process(es)* which is connected with the computer via *sensors* and *effectors*, input from user(s), and output to user(s). The user is supervising the presentation of the system status and preparing to program a new task via diverse input devices. The system is responsible for:

- Capturing the incoming data from the manipulation devices of user and *sensors*.
- Deducing the control decisions in terms of the captured data from user and *sensors* and integrated knowledge bases about the application domain.
- Producing commands to initiate *effectors* to execute the control decisions.
- Presenting the system status to the user for further interaction.



**Fig. 2: A supervisory interactive computer system**

Shortly, in supervisory interactive computer systems, *supervisory control* implies two things:

- *Users*

As in automation and control systems, users still need to take the responsibilities of planning, programming, monitoring, and intervening.

- *User interfaces*

The interfaces need to have the ability to observe and leverage the *controlled processes* and deduce the optimal dialogue possibilities to their users.

## 2.2.2 What is a WIMP<sup>+</sup> User Interface?

User interfaces in supervisory interactive computer systems can automatically observe the *controlled processes* by capturing data, both continuous and discrete, from *sensors* that are attached to them. They can acquire input from users, too. Their output involve both the part of system presentation for users' interaction and the part for initiating *effectors* to leverage the *controlled processes*.

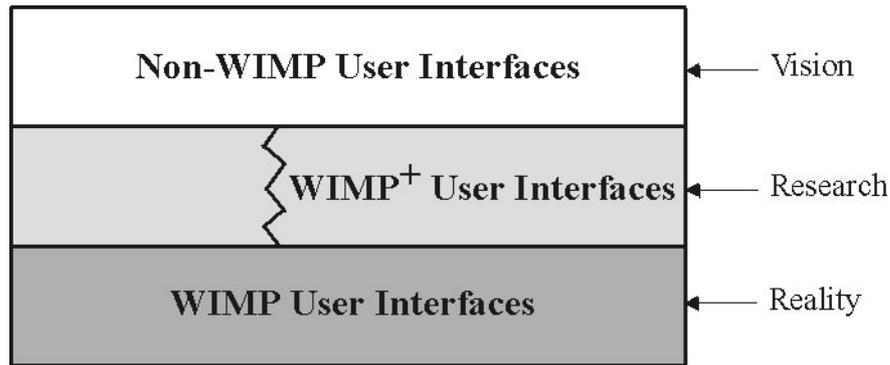
Therefore, in addition to the input and output of WIMP user interfaces, user interfaces in supervisory interactive computer systems could deal with continuous, parallel, real-time, and high bandwidth input and output. Diverse *sensors* give the ability to the user interfaces automatically observing the *controlled processes*; diverse *effectors*, on the other hand, equip them with the ability to leverage the *controlled processes*; adequate knowledge base can be integrated for deducing optimal dialogue sequences and control decisions; richer and suitable new interaction techniques and devices can be utilized for HCI as well.

*Obviously, user interfaces in supervisory interactive computer systems have already gone beyond the original WIMP metaphor! They can provide the function of observing and leveraging the controlled processes to fulfill users' tasks, which is one of the major functions of Non-WIMP user interfaces!*

*However, it does not mean to build such user interfaces, all WIMP interaction techniques and devices should be thrown away. Instead, some of them could still be inherited.*

*In order to distinguish the user interfaces in supervisory interactive computer systems from WIMP user interfaces and Non-WIMP user interfaces, hence, in this thesis, the term WIMP<sup>+</sup> user interfaces is used [130].*

In future, *WIMP<sup>+</sup> user interfaces* will include other substantial functions of Non-WIMP user interfaces. But at present, *WIMP<sup>+</sup> user interfaces* would be any interface based on, but not limited to, the interaction techniques and devices of windows, icons, menu, and pointing device: the mouse, for supervisory control. They could be regarded as a kind of future user interfaces which are towards Non-WIMP user interfaces, as Fig. 3 depicts.



**Fig. 3: WIMP<sup>+</sup> user interfaces inherit partially WIMP interaction techniques and devices to provide supervisory control function and as many other functions of Non-WIMP user interfaces as possible**

## 2.3 Examples

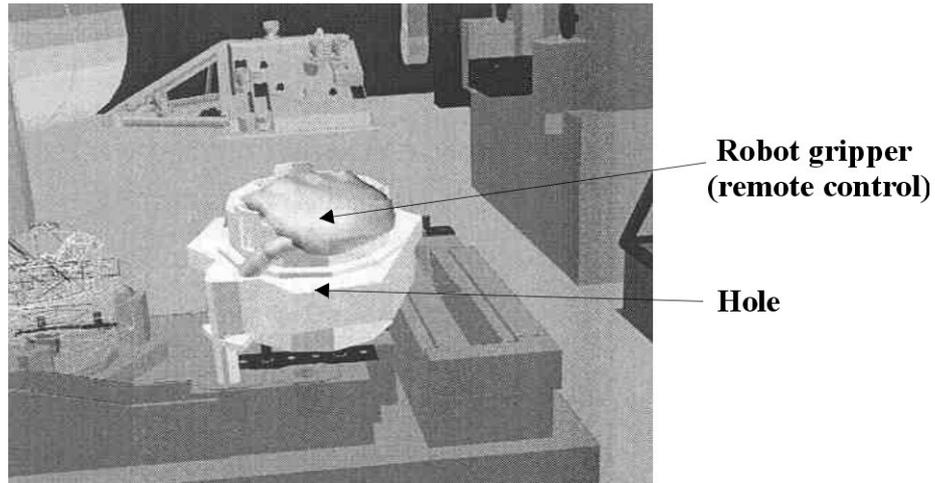
### 2.3.1 User Interfaces for Robot Control

To aid users control a robot<sup>5</sup>, especially for telecontrol, one of the most effective methods is to create a user interface which maps the movable parts of the robot and its task environment in a 2-D or a 3-D form. Then let its users control the real world robot to fulfil tasks via manipulating the related representation of the robot parts in the user interface.

For example, Fig. 4 shows a user interface for telecontrol a space robot from the earth. The user interface maps the gripper of the robot and its task environment in a 3-D virtual workspace. By manipulating the representation of the robot gripper in the interface, a user can control the gripper of the real world robot to put a tool in a hole in space [ 16].

---

<sup>5</sup> In this thesis, robot refers to non-autonomous robot.

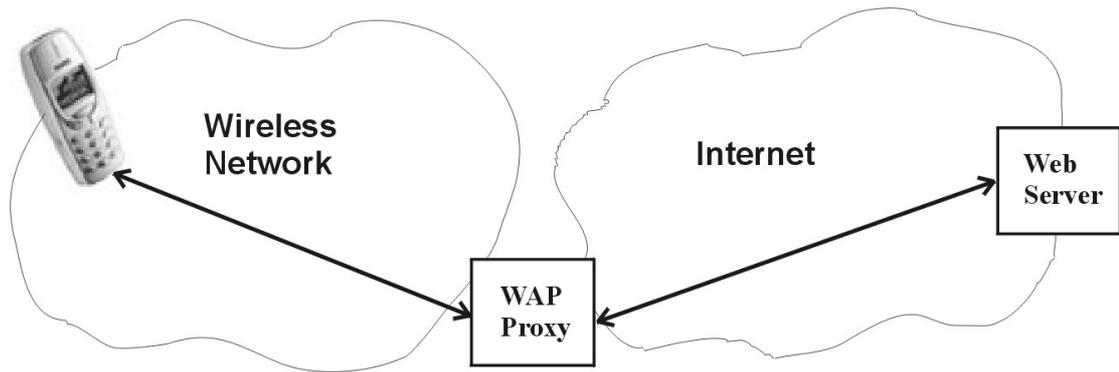


**Fig. 4: A user interface for telecontrolling a space robot, which maps the robot gripper and its task environment in a virtual workspace [16]. Obviously, it is no more a WIMP user interface!**

Like this example, *many user interfaces for robot control are no more WIMP user interfaces!* They deal with *sensors, effectors, users' input, and system presentation*. They can automatically observe and leverage *controlled processes* as design. Obviously, they meet the definition of *WIMP<sup>+</sup> user interfaces!*

### **2.3.2 User Interfaces of WAP Mobile Phones**

WAP mobile phones, by use of micro browsers, can bring the Internet contents to their users. As Fig. 5 shows, with a WAP mobile phone, in order to access the Internet a user enters commands to its micro browser. The browser receives the user's commands, analyzes them, translates the commands as the WAP requests and transfers the requests to a WAP proxy. The WAP proxy receives the WAP requests, translates them as the WWW requests and then transmits the requests to the corresponding WWW server. After the WWW server responds, it sends the requested content as WWW data back to the WAP proxy. The WAP proxy then interprets the content to the WAP data, transmits the data further to the phone via wireless network. The browser then displays the requested Web content in its screen. Additionally, a mobile phone is used to fulfil other telephony services, too.



**Fig. 5: A WAP wireless phone accesses a WWW Web page**

The interface of a mobile phone usually occupies the following features:

- Observe and leverage the task environment. For example, the user interface has to sense if any call is coming, and initiate the telephony service if applicable.
- Parallel and continuous input and output. For example, it is possible that the Web data from the Internet and a telephony call are arriving parallel to a mobile phone (Both are continuous data.).
- Innovative interaction devices. Instead of the mouse and conventional keyboard devices, there is other interaction devices used in a mobile phone, e.g., mini display, on-screen keyboard, character recognition screen, vibration, etc.

Fig. 6 shows a WAP mobile phone from Ericsson [72]. Like this WAP mobile phone, *the user interfaces of other mobile phones are no more WIMP* due to the above features. However, they could be treated as *WIMP<sup>+</sup> user interfaces!*



**Fig. 6: The user interface of the Ericsson R380 [72]. It has already gone beyond WIMP metaphor!**

### **2.3.3 User Interfaces of Car Driver Assistant Systems**

Car driver assistant system is one of the key projects being carried out in several world leading technology automobile companies<sup>6</sup>. Their goal is to improve the safety of steering. Traditionally, drivers steer automobiles mechanically according to their experiences and what they see. Personnel experiences and healthy condition of drivers decide mainly the safety. The longer time drivers are in steering, the more fatigue they are. And fatigue is one of the main reasons for error judgement and operation. Car driver assistant systems are a long time dream of human beings.

Fortunately, with today's electronic and computer technology, many components of a vehicle can be controlled by wire, which makes the development of car driver assistant systems possible.

A car driver assistant system is to help a driver monitoring the safety of a vehicle at all times. The computer receives the driver' s commands and interprets them as demands for a particular driving status, i.e., accelerating, braking, steering, reversing, then decides in what way these commands can be executed most effectively and safely, e.g., the turning angle of the wheels for cornering or the engine speed for starting off on an icy road. The decision is dependent on the driving status, e.g., road condition, wheel and engine revolutions, and bodywork movements, which is recorded by various *sensors* in time. Since the computer knows all the vehicle's technical data and current driving conditions, the driver can fully exploit the vehicle' s performance potential without exceeding the physical limits.

Fig. 7 shows a user interface of a driver assistant system prototype from DaimlerChrysler AG [122]. *The user interface has gone beyond the WIMP metaphor!* Yet, it could be treated as a *WIMP<sup>+</sup> user interface!*

---

<sup>6</sup> For example, DaimlerChrysler AG plans to spend 2,2 billion Euro into their development within these three years [122].



**Fig. 7:** In a commercial vehicle prototype of DaimlerChrysler AG, an innovative user interface for steering has taken over the conventional one, where a display that replaces the conventional dashboard can provide richer information, and two side-sticks that replace a steering wheel and pedals are more safe, simple, and convenient [122]. This user interface is no more WIMP!

## 2.4 Characteristics of WIMP<sup>+</sup> User Interfaces

In addition to the above examples, there are many similar user interfaces that have already gone beyond the WIMP metaphor but could be treated as *WIMP<sup>+</sup> user interfaces*. For example, user interfaces in plant automation and control, user interfaces in email filter applications, user interfaces in truck logistics and maintenance, user interfaces in distributed multi-user database systems, user interfaces in automation rail systems, and user interfaces in aviation, etc.

Compared with traditional WIMP user interfaces, *WIMP<sup>+</sup> user interfaces* occupy the following outstanding features:

- *Manifold interaction devices and techniques*

In addition to windows, icons, menus and pointing device: the mouse, all available interaction devices and techniques can be exploited in *WIMP<sup>+</sup> user interfaces*, especially those natural interaction techniques such as two-handed manipulation.

- *Higher bandwidth and parallel input/output*

It is due to simultaneously using diverse interaction devices that deal with high bandwidth signals.

- *Continuous and real-time response and feedback*

Unlike the syntactic command based user interfaces, *WIMP<sup>+</sup> user interfaces* need to respond both continuous and discrete input, and need to initiate *effectors* to leverage *controlled processes* in time, too.

- *Ubiquitous computing*

Unlike a WIMP user interface, a *WIMP<sup>+</sup> user interface* can possibly be split into several parts and different parts could be embedded in different places.

**Table 2: The features of WIMP user interfaces, WIMP<sup>+</sup> user interfaces, and Non-WIMP user interfaces**

	WIMP	WIMP <sup>+</sup>	Non-WIMP
High Bandwidth Input and Output	Non	Maybe	Yes
Many Degrees of Freedom	Non	Maybe	Yes
Real-Time and Parallel Interaction	Non	Yes	Yes
Continuous Response and Feedback	Non	Yes	Yes
Natural Languages	Non	Maybe	Yes
Agents	Non	Maybe	Yes

As Table 2 shown, *WIMP<sup>+</sup> user interfaces* possess at least more than one outstanding feature of Non-WIMP user interfaces [40] which have never been dealt with within traditional WIMP user interfaces. And they may possess as many features of Non-WIMP user interfaces as possible in the future.

## **2.5 Summary**

In this chapter, the evolution of user interfaces was surveyed briefly, from the traditional one: first generation user interfaces, second generation user interfaces, third generation user interfaces (WIMP user interfaces), to the future one: Non-WIMP user interfaces. From the study, it is clear that the drawbacks of each user interface generation, the lack of supporting application requirements in that time, and the hardware technology improvement are the three main reasons of the birth of new user interface generation. Based on analyzing the problems existing in traditional WIMP user interface and investigating the new technology possibility, four innovative functions of Non-WIMP user interfaces were deduced: 1) more powerful and natural languages for users interacting with computers; 2) richer interaction devices and techniques; 3) agents; and 4) supervisory control. And to investigate the user interfaces for supervisory control is chosen as the research topic of this thesis.

After elaborating on the implication of supervisory control in automation and control systems, supervisory interactive computer systems and *WIMP<sup>+</sup> user interfaces* were defined. To expound *WIMP<sup>+</sup> user interfaces* more intuitive, three examples were given. Finally, by comparing *WIMP<sup>+</sup> user interfaces* with WIMP user interfaces and Non-WIMP user interfaces, the features of *WIMP<sup>+</sup> user interfaces* were further studied.

This chapter is about the basic concepts of this thesis. In the next chapter, the state-of-the-art models and tools for *WIMP<sup>+</sup> user interfaces* will be elaborated.

## **3 The State-of-the-art Models and Tools for WIMP<sup>+</sup> User Interfaces**

### **3.1 User Interface Models**

#### **3.1.1 High Level Models**

A model is a representation of a design or the idea of a design. Usually, it contains a set of rules and plans. In software engineering, several models have been used for software projects planning and control [9] [34] [62], e.g., the waterfall model, the spiral model, etc. As the user interface part of a software is getting more and more large and complex, user interface has become an independent part to be researched, analyzed, designed, implemented, tested, and evaluated. As a result, an engineering discipline, user interface engineering, is emerging to control the user interface development process in order to guarantee the quality [27].

According to [98], the essential goals of user interface engineering are:

- Define the proper functionality of a user interface.
- Ensure the reliability, availability, security, and data complete.
- Consider the standardization, integration, consistency, and portability.
- Keep in schedules and budgets.

To reach these goals, several models or methodologies have been developed, e.g., life cycle model [65], usability engineering [81], and the new emerging user-centered design methodology [109]. In this thesis, these models or methodologies are called *high level models* while

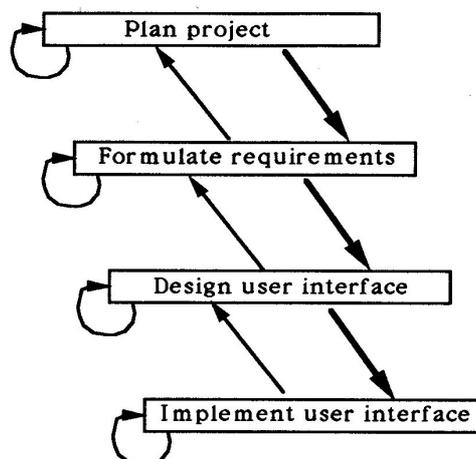
others used in a concrete development phase are called *low level models*, e.g., task models, HCI models, software models, test models, etc.

### 3.1.1.1 Life Cycle Model

In software engineering one of the most widely used life cycle models is the waterfall model. It divides the software development process into several phases [9]. All the phases are executed properly with controllable criteria in terms of a defined flow: the next phase begins only after the current phase finishes with several elaborated documents. In user interface engineering it is also used to for controlling the development process.

But user interface development involves human factors and its development is based on large amount of iterations of evaluation, testing, and assessment. Hence, the original waterfall model can not work well for user interface realization since it is short of consideration of these iterations. Obviously, it needs to be modified.

A variant waterfall model [65] with consideration of the iteration of plan, requirement, design, and implementation for user interface development is shown in Fig. 8. It depicts both the waterfall nature that the completion of one phase leads to the beginning of the next one (with the heavy arrows) and the iteration within each phase (with the light semicircle arrows) or feedback between phases (with the light arrows).



**Fig. 8: A variant waterfall model for user interface development process [65]**

Although this model clearly defines the obligation of individuals in a development team<sup>7</sup>, however, as Boehm points out: “*Document-driven standards have pushed many projects to write elaborate specifications of poorly understood user interfaces and decision-support functions, following by the design and development of large quantities of unusable code.*” [9, p63] Moreover, it is short of consideration of end users’ involvement and usability issues.

### **3.1.1.2 Usability Engineering**

With combination of other disciplines dealt with human factor, e.g., psychology, Jakob Nielsen [81] develops usability engineering methodology. The usability encompasses five essential aspects: easy to learn, efficient to use, easy to remember, few errors, and subjectively pleasing [81, p25]. To achieve the usability, 11 phases should be followed: “1) *Know the user*; 2) *Competitive analysis*; 3) *Setting usability goals*; 4) *Parallel design*; 5) *Participatory design*; 6) *coordinated design of the total interface*; 7) *Apply guidelines and heuristic analysis*; 8) *Prototyping*; 9) *Empirical testing*; 10) *Iterative design*; 11) *Collect feedback from field use.*” [81, chapter 4]

Compared with the waterfall model, usability engineering is less formal, e.g., the obligation of individuals of a development team are not clearly assigned, and how to set and measure the quality of usability are also indistinct.

### **3.1.1.3 User-Centered Design Methodology**

No doubt, completely understanding and acquiring the end users’ needs should be the first and foremost step for user interface development. However, it is difficult for a development team to completely “*know the user*” [81] only by interview, meetings, and visiting.

---

<sup>7</sup> For example, system analysts are responsible for understanding the end users and establishment of conceptual interface model while user interface designers are responsible for translating the conceptual model into a design model suitable for human computer interaction, and programmers realize the design model.

*“...there are (in the business area as well as in the engineering area) situations ... where prospective users are not able to articulate their needs or where the needs are of a too much subjective nature...” [51, p943].*

Indeed, during two industry projects [126] [129] the thesis author has experienced similar situations: at the beginning of a project, some prospective users can not clearly declare their requirements.

To articulate the needs of end users, end users' active involvement is strongly required throughout a project. Accordingly, other methodologies, which aim to activate end users' involvement have been recently investigated in both communities of software and HCI. And the most well-known one is user-centered design methodology (e.g., [109] [98, p104]).

Compared with life cycle model and usability engineering methodology, user-centered design methodology has two outstanding features. First, it calls for an interdisciplinary development team, which comprises system analysts, user interface designers, programmers, usability specialists, marketing specialists, and other experts in terms of the project's need. Second, it imposes the interdisciplinary team to work with end users through out the whole development process. By involving end users in each development phase, the interdisciplinary team can timely obtain the end users' reaction, feedback, and comments. Consequently, it helps to achieve the usability user interfaces more effectively and efficiently.

But since user-centered design methodology is a new emerging methodology, it is still short of rules to control the concrete development process. Therefore, in practice, the concrete executive details of user-centered design methodology could be varied from team to team. For example, the User-Centered Design methodology of IBM [109] is not the same as that of the Logical User-Centered Interactive Design Methodology [98, p104] in detail. Nevertheless, both methodologies possess the above two features.

## **3.1.2 Design Patterns: a Powerful Modeling Methodology**

### **3.1.2.1 The Context of User Interface Modeling Methodology**

High level model is for controlling user interface development process, while low level models, e.g., task models which describes users' tasks, HCI models for specifying the human and computer interaction, software models for describing the software architectures, test models for usability evaluation, etc., are used to support a concrete development phase, respectively. And since the trend of user interface development involves increasingly large amount of cooperation work among the interdisciplinary development team which includes the end users, system analysts, user interface designers, programmers, usability specialists, marketing specialists, and other experts, each may occupy different knowledge background and technique expertise, modeling methodologies for specifying and documenting low level models are extremely critical.

Indeed, a powerful modeling methodology can help the interdisciplinary team members to clearly and easily present their ideas. On the contrary, a non-powerful one could be adhered to a set of complex rules that are difficult to be mastered or difficult for people to express their ideas clearly. If a model is depicted with the non-powerful methodology, it could be ambiguous, difficult to understand, or short of logic, and consequently impede the communication among the interdisciplinary team.

In software engineering, there are two kinds of widely accepted modeling methodologies. One is the structured methodology developed within 1960s and 1970s [9] [34] [62], e.g., Jackson Structured Design methodology and Hierarchical plus Input and Output charts<sup>8</sup>. The other is the modern object-oriented methodology which has been developed since early 1980s [11] which, by applying the Unified Modeling Language (*UML*) [10], both the attributes and behaviors of software can be clearly specified. These two kinds of modeling methodologies have been widely

---

<sup>8</sup> An introduction of Jackson Structured Design Methodology and Hierarchical plus Input and Output charts can be found in [34] [62].

used individually or in mixture to guide development of large amount of software projects (e.g., [126] [129]).

But for specifying the user interface part, no formal or semiformal methodology has been widely disseminated so far [34, p29] [52] [98, p54]. First, it is due to the inherent complexity of user interfaces which deal with the field of human computer interaction that cuts across several disciplines [1]. Second, developers had been blinded by the fact that a user interface is a part of an application for a long time<sup>9</sup>, which causes the fact that its inherent complexity was not been properly attacked. Third, the dramatic change of two subjects of user interfaces (see section 2.1.2 of this thesis for details), computing capability and the users' requirements, makes the user interface specification and documentation difficult to stick to any formal methodology.

However, user interface researchers have never stopped their research works. Totally, user interface modeling methodologies can be divided into two kinds. One is the non-formal methodology, e.g., natural languages like Chinese, English, German, etc. With substantial vocabularies and sentences, a natural language is the easiest way for people to describe their ideas and to communicate. But a natural language is usually ambiguous. The other one is the formal or semiformal methodology like Grammars, State Transition Diagrams, and Event Languages [35] [41] [57] [98]. Although these methodologies have overcome the ambiguity of natural languages, they are less intuitive, complex, and professional people oriented. Moreover, they are not optimal for specifying and documenting all aspects of a user interface.

Recently, both software community and HCI community have been attracted by *the design pattern methodology*. The annual Pattern Language of Programs (*PLoP*) conference (held in USA since 1994) and annual EuroPLoP conference (held in Germany since 1996), which are two major forums for standardizing the new discovered software patterns [19], have obtained widely attention in both communities. And several HCI pattern workshops have also been held

---

<sup>9</sup> It was until 1980s, especially after the first CHI conference, which was held in March 1982, in Gaithersburg, Maryland, USA, user interface development has been formally considered as an independent field [27].

to activate HCI design pattern research, e.g., Patterns Languages for Interaction Design: Building Momentum [42], and recently Patterns in Human Computer Interaction [85].

### **3.1.2.2 The Root of Design Patterns**

The theory of design patterns is invented by architect Christopher Alexander [2] [3]. The Alexanderian design patterns aim at guiding inhabitants to shape their rooms, houses, streets, and towns themselves in “*the timeless way*” [3]. The essences of “*the timeless way*” are “*The Quality; The Gate; and The Way.*” “*The Quality*” means the root criterion of life and spirit in a man, a town, a building, or a wilderness; it is objective and precise, but it cannot be named [3]. This is the so-called “*The Quality Without A Name*”. “*The Gate*” is actually a established *living* pattern language. A *living* pattern language gives each person who uses it the power to create an infinite variety of new and unique *living* buildings. By using of it people can reach the quality. “*The Way*” is about the process that people use a *living* pattern language in their construction practice to pursuit the timeless way.

According to Alexander, a pattern language is generated from a pattern system which contains two sets [3]: patterns and rules for their combination.

*“...Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, ...”* [2, page “x” in introduction]

Moreover, no pattern exists isolated: a pattern must be coexist with other patterns; it can be used to support the larger patterns; and it can be supported by the smaller patterns.

Alexanderian pattern language contains 253 patterns [2]. Fig. 9 is a sample pattern from the language. As this sample pattern, all these 253 patterns encompass the following eight elements:

- Name: defines what a design pattern is called. It conveys the essence of a pattern succinctly.
- Picture: shows an archetypal example of a pattern. Alexander and his colleagues use a photograph recording the real world example.

I 74 TRELLISED WALK\*\*



809

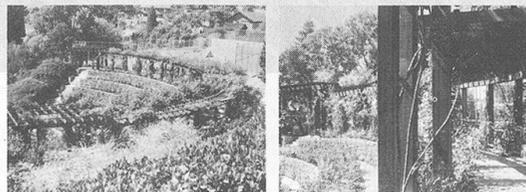
... suppose the main spots of the garden have been defined—OUTDOOR ROOM (163), TREE PLACES (171), GREENHOUSE (175), FRUIT TREES (170). Now, where there is a special need to emphasize a path—PATHS AND GOALS (120)—or, even more important, where the edges between two parts of a garden need to be marked without making a wall, an open trellised walk which can enclose space, is required. Above all, these trellised walks help to form the POSITIVE OUTDOOR SPACES (106) in a garden or a park; and may perhaps help to form an ENTRANCE TRANSITION (112).



Trellised walks have their own special beauty. They are so unique, so different from other ways of shaping a path, that they are almost archetypal.

In PATH SHAPE (121), we have described the need for outdoor paths to have a shape, like rooms. In POSITIVE OUTDOOR SPACE (106), we have explained the need for larger outdoor areas to have positive shape. A trellised walk does both. It makes it possible to implement both these patterns at the same time—simply and elegantly. But it does it in such a fundamental way that we have decided to treat it as a separate pattern; and we shall try to define the places where a trellised structure over a path is appropriate.

1. Use it to emphasize the path it covers, and to set off one part of the path as a special section of a longer path in order to make it an especially nice and inviting place to walk.



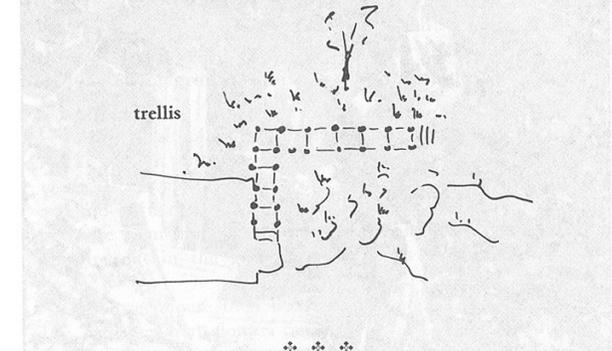
A trellis gives shape to an outdoor area.

810

I 74 TRELLISED WALK

2. Since the trellised path creates enclosure around the spaces which it bounds, use it to create a virtual wall to define an outdoor space. For example, a trellised walk can form an enormous outdoor room by surrounding, or partially surrounding, a garden. Therefore:

Where paths need special protection or where they need some intimacy, build a trellis over the path and plant it with climbing flowers. Use the trellis to help shape the outdoor spaces on either side of it.



Think about the columns that support the trellis as themselves capable of creating places—seats, bird feeders—COLUMN PLACES (226). Pave the path with loosely set stones—PAVING WITH CRACKS BETWEEN THE STONES (247). Use climbing plants and a fine trellis work to create the special quality of soft, filtered light underneath the trellis—FILTERED LIGHT (238), CLIMBING PLANTS (246). . . .

811

Fig. 9: A sample Alexanderian design pattern

- Context: explains the connection of a pattern with other patterns.
- Bold type problem headline: describes the problem of a pattern in short.
- Body of the problem: describes the evidences of a pattern.
- Bold type solution headline: describes the solution to the stated problem within the stated context.
- Sketch: shows the solution in an abstract and intuitive way.
- Reference paragraph: links smaller relevant patterns to complete the described pattern.

Besides, every Alexanderian design pattern possesses a ranking for validity. It follows directly the name of a pattern. It is expressed by none, one, or two asterisks, which states low, general, and high confidence of the authors to a pattern. Additionally, two three diamonds symbols are used to indicate the main body of a pattern. The first one which situates between Context and Bold type problem headline marks the beginning of the main body of a pattern. And the second one which locates between Sketch and Reference shows its end.

Since every pattern is denoted in the same form, since the form itself is very rational with problem/solution pair and intuitive with picture and sketch, it is quite understandable. Moreover, all these 253 design patterns are linked together in terms of rules which makes them as a whole. This in turn makes a very powerful *living* pattern language. By using the language, nonprofessional inhabits can communicate with professional architects. Even more, they can build their houses and towns of their own since the language captures not only designs but also how to design.

The great contribution of Alexander and his colleagues is not just the creation of a pattern language for architecture<sup>10</sup>, but it is his invention of a universal modeling methodology to

---

<sup>10</sup> Actually, few architects have used Alexanderian pattern language to construct building and towns up till now [15].

valuable systems: *living* buildings. The methodology leads to create a common vocabulary, i.e., a set of patterns, for expressing the concepts of a discipline, and a language to relate them together. This is fundamental to other science and engineering disciplines including software engineering, HCI, and application domains.

### 3.1.2.3 Design Patterns in Software Engineering

Influenced by the Alexanderian design pattern theory, Kent Beck and Ward Cunningham used design patterns guiding non-Smalltalk programmers using Smalltalk to construct user interfaces in an experiment. The results of the experiment were reported at OOPSLA1987 in Orlando, USA [6]. In 1991, James O. Coplien published a book which deals with a pattern style: Idioms [21]. But it is until the Gang of Four (*GoF*<sup>11</sup>) book [36] which was published in 1995 that software patterns became popular world wide.

*“A design pattern is a description of communication objects and classes that are customized to solve a general design problem in a particular context.”* [36, p23]

In the spirit of Alexander, the primary goal of software patterns is also to model sound software architectures and designs [26]. And it is because that software patterns touch critical issues central to software development they are so attractive and widely disseminated [22]. But since a software pattern is just a model, it depends on the insight of the people who creates and uses it. Nevertheless, since mature engineering disciplines have handbooks (e.g., [14]) which include successful solutions to known problems, software patterns hopefully constitute a basis for handbooks of software engineering (e.g., [17] [20] [36] [70] [92] [93] [115]).

To document a software pattern, there are several forms: the Alexanderian form, the GoF form, the Portland form, and the Coplien form [22]. Although different forms contain different elements, the common elements of a software pattern are [22]:

---

<sup>11</sup> GoF refers to the authors of “*Design Patterns: Elements of Reusable Object-Oriented Software*” [36]: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

- Name: as the name of Alexanderian patterns, it conveys the essence of a pattern succinctly. A name is important to a software pattern since it quickly becomes part of the design team vocabulary; and it will be one of the first things a designer encounters when seeking a solution.
- Intent: summarizes what a pattern does.
- Context: describes the background of a pattern. It specifies the relationship of a pattern with others.
- Problem: describes the problem to be solved. An appropriate and concise problem statement helps the designers to decide if a pattern deals with their problems.
- Forces: describes the conflict sides of a pattern in detail. The term Forces comes from Alexanderian patterns. *“A building architect designs arches and walls to balance the forces of gravity with the forces from adjoining structures, so the structure is balanced and centered.”* [22]
- Solution: solves the stated problem. A good solution should be detailed enough to tell the designer what to do and how to do it. And it should be general enough to address the stated context.
- Sketch: conveys the structure of the solution to the problem in an intuitive and abstract way, e.g., the UML diagram.
- Resulting Context: concludes the benefits and new problems that a pattern brings.

In addition to Name, Intent, Context, Example, Problem, Forces, Solution, and Resulting Context, the following elements occasionally used also by other authors will be encompassed in the software pattern documentation of this thesis (see chapter 4 for details):

- Dynamic: describes the run-time behavior of a pattern by examples.
- Implementation: gives a sample implementation strategy to demonstrate how to implement a pattern.

- Variant: what patterns are the variants of a pattern.
- Known uses: in what applications has a pattern been successfully applied?
- Related patterns: introduces the smaller patterns that a pattern contains.
- Acknowledgement: acknowledge to the peoples who contribute to a pattern.

Here, Dynamic are split from the Solution in general form in order to clearly describe the run-time behavior of a pattern; Implementation illustrates a sample implementation strategy independent of the abstract solution; and the Sketch in general form is distributed throughout the elements of Example, Solution, Dynamic, and Implementation.

### 3.1.2.3.1 The Scale of Software Patterns

As design patterns in architecture, software patterns cover also various ranges of scale and abstraction. A software pattern can be used to support other larger patterns; and it can be built with other smaller patterns. According to the range of scale, software patterns can be divided into three categories: architectural patterns, element patterns (design patterns), and idioms [ 17].

Architectural patterns are on the highest level of software patterns. It is actually a system of patterns used to capture the system structure of a family of applications. It is built up with a set of element patterns. It provides a set of components (or subsystems) with appropriate functionalities and includes rules for organizing them together. It is fundamental to an application since it determines the quality criteria such as robustness, flexibility, maintainability, and other system wide structural properties.

*Actually, a software architectural pattern could be seen as a pattern language for construction of a family of similar applications since it involves a set of element patterns and defines clearly their locations for building up the applications.*

Element patterns are in the medium-scale level. They are smaller in scale than architectural patterns. And they are independent of a particular programming language. They are used to support their high level patterns, in this case, the architectural patterns. The most well-known 23 element patterns which deal with creation, structure, or behavior of objects are documented in [36].

An idiom is a low-level pattern. It is specific to a programming language. It describes how to implement particular aspects of components or the relationships between them with the given language. Different programming languages could provide different idioms for an element pattern.

### **3.1.2.4 Design Patterns in HCI**

Inspired by the fact that software community has successfully adopted the design pattern methodology in modeling software architecture and designs, HCI community started to intensively explore the utility of the pattern methodology in HCI since CHI1997. HCI pattern researchers have met each other in CHI1997 [5], Interact1999 [43], CHI2000 [42], and IFIP 13.2 Workshop on HCI Patterns [85], respectively, to exchange their insights and findings. They believe that the field of HCI is closer to architecture than software engineering since the goal of HCI is also to construct a kind of environment for people. Several HCI design patterns and pattern languages have been developed (e.g., [12] [108] [110]). However, as discussed at [85], due to the inherent complexity of HCI, up till now there are still lots of questions that disturb HCI pattern researchers. For example,

- The definition of HCI patterns and pattern languages. HCI patterns and pattern languages try to embrace all the aspects of what end users touch. And from end users' point of view, a user interface is composed of elements of interaction tasks, interaction techniques, and interaction devices. The manifold interaction tasks, interaction techniques, and interaction devices result in the present manifold genres of HCI patterns and pattern languages that don't fit together.
- The form of HCI patterns. Since HCI deals with multiple disciplines, from the different disciplinary point of view different HCI pattern author uses different forms. For example, someone uses software pattern form (e.g., [133]), someone prefers the Alexanderian form (e.g., [12]), and someone adds the usability aspect in the description (e.g., [110]).
- The presentation of the dynamic aspect of a HCI pattern. Unlike architecture and software, a user interface often deals with a temporal and dynamic dimension that is

harder to represent in the traditional pattern form. It is even more difficult to find a universal methodology for representing this temporal and dynamic aspect.

- The categories of HCI patterns and pattern languages. Several organising principles are competing for categorizing HCI patterns and pattern languages, e.g., the scale organising principle (e.g., [12]), the organising principle related to interaction tasks (e.g., [33]), etc.
- The interlink of HCI patterns and pattern languages. Most HCI patterns and pattern languages are kept by their authors. Only some of them have been submitted to HCI pattern workshops for discussion; or have been interlinked together by limited Web pages (e.g., [106] [107] [108]).
- The dissemination of HCI patterns in the development practice. The present distribution state of HCI patterns and pattern languages obstructs the developers to find a relevant HCI pattern, which in turns make it difficult to disseminate the valuable patterns for supporting the development practice.
- The possibility of end users as designers. One goal of Alexanderian pattern language is to let inhabitants design their house as architects. Several HCI pattern researchers argue that HCI pattern languages should inherit Alexander's legacy. Others, on the contrary, based on their design experience, insist that end users could not really design artifacts of their own. Instead, they just evaluate what have designed for them. Nevertheless, HCI patterns and pattern languages should be readable and understandable for all individuals within the HCI interdisciplinary development team since the goal of HCI patterns is to provide a common ground [108] or lingua franca [30] for all members of a HCI interdisciplinary team to work together.

Fortunately, at the Workshop of Pattern Languages for Interaction Design: Building Momentum [42] at CHI2000, at which the thesis author was present, the definition of HCI pattern and HCI pattern form were established:

*“A HCI design pattern captures the essence of a successful solution to a recurring usability problem in interactive systems.”*

A HCI pattern should consist of the following elements:

- **Name:** like Alexanderian patterns and software patterns, the name of a HCI pattern is critical, too. It should also be able to convey the essence of a HCI pattern succinctly.
- **Ranking:** indicates the validity of a pattern and its authors' confidence to it. Ranking is borrowed from the Alexanderian patterns. None, one, or two asterisks can be used to state low, general, and high confidence of an author to a pattern.
- **Example:** represents the archetype of a HCI pattern. It could be a photography, a film, or a text description.
- **Context:** similar to the Context of Alexanderian patterns and software patterns.
- **Problem:** describes the problem to be solved; states the contradictions or the forces of a family of interactive systems before the stated HCI pattern is applied.
- **Evidences:** gives the rationale why the stated pattern is necessary.
- **Solution:** tells the reader how to solve the stated problem.
- **Sketch:** extracts the solution in an abstract and intuitive way.
- **References:** like the Reference paragraph of Alexanderian patterns, it relates other smaller patterns with which the stated pattern is built upon.

Alexanderian patterns and pattern language take about 10 years to go its way, from the emerging state, to a mature theory, and finally used in practice. Software patterns have also experienced almost 15 years. Compared with architecture and software engineering, HCI pattern research is still in the beginning state. Moreover, Alexanderian patterns and pattern language are based on thousands of years building practice of people, software patterns have also a relative long and stable background, while the base of HCI pattern is quite temporal and dynamic. Obviously, HCI pattern researchers are confronted with more challenge.

### 3.1.2.5 Application Domain Patterns

To achieve an interface specific to end users' tasks, user interface designers need a kind of models, the abstract user interface models of end users, which extract the tasks of end users and knowledge of application domain. But user interface designers are not application domain experts. It is difficult for them to understand the users' tasks, give better solutions to fulfil the tasks is even a thorny problem. As [76] points out, in addition to the difficulties associated with designing any complex software system, user interface designers are obstructed by following problems:

- “• *Designers have difficulty thinking like users;*
- *Tasks and domains are complex;*
- *Various aspects of the design must be balanced (standards, graphic design, technical writing, internationalization, performance, multiple levels of detail, social factors, legal issues, and implementation time);*
- *Existing theories and guidelines are not sufficient;*
- *Iterative design is difficult.*”

And the difference between designers and users is the first problem while the complexity of tasks and domains is located in the second. These two problems indicate the importance of the concept models of end users for user interface development.

On the other hand, domain experts do possess expertise and relative mature abstract user interface models related to their application domains. They master the essences of the tasks they are doing, e.g., the objects relevant to the task, relationships among these objects, and operations needed by these objects. They are proficient at the present ways they are using to fulfill the tasks. They also know the constraints within the present systems and the reasons why they need to change the existing task fulfillment ways. Furthermore, they usually have visions of new systems. To aid user interface designers overcome the two above problems, it could be very helpful to specify the abstract user interface models of domain experts in a formal or semiformal format, e.g., using the design pattern methodology to specify them as application domain patterns such as [12] [133].

An application domain pattern could be users' task oriented. From the thesis author's point of view, it could include the following elements:

- Name: defines what this application domain pattern is called. It should imply the essence of a task succinctly.
- Intent: summarizes about what is this application domain pattern.
- Context: describes the background of this application domain pattern.
- Example: represents the archetype of the task that this application domain pattern describes.
- Problem: describes the problem of the existing user interface.
- Forces: describes the conflict side of the problem in detail.
- Solution: outlines the vision of how to solve the stated problem in the envisioned user interface. It could be either detailed or abstract.
- Domain model: depicts the domain knowledge needed in the solution. It could include task related objects, relationships among these objects, operations needed by these objects. It could include complex algorithms like a kinematical model of a robot, Fourier Expansion equations or very simple operations like access or assignment a data.
- Sketch: conveys the structure of the solution in an intuitive way.
- Known uses: likes the Known uses element of software pattern, it gives the evidences to the solution, i.e., in what applications has the solution been successfully used.
- References: describes the relationships among this application domain pattern and other application domain patterns. For example, if an application domain pattern is used to describe a complex composite users' task, it could embrace several smaller application domain patterns, each of which describes a related subtask. But if an application domain pattern is used to describe a very simple task, it could contain no smaller application domain pattern.

The use of application domain patterns in user interface development practice has been explored in several projects (e.g., [12] [39] [132]). The result is promising. However, its validity needs to be further examined.

### 3.1.3 Hot-UCDP: A Framework for User-Centered Design Methodology

As already described in section 3.1.1.3, user-centered design methodology is one of the most promising high level models for user interface realization. It emphasizes the importance of end users' participation in the user interface development process. Its goal is to understand and acquire users' needs timely. However, as section 3.1.1.3 points out, since user-centered design methodology is a new emerging methodology, it is still short of rules to control the concrete development process.

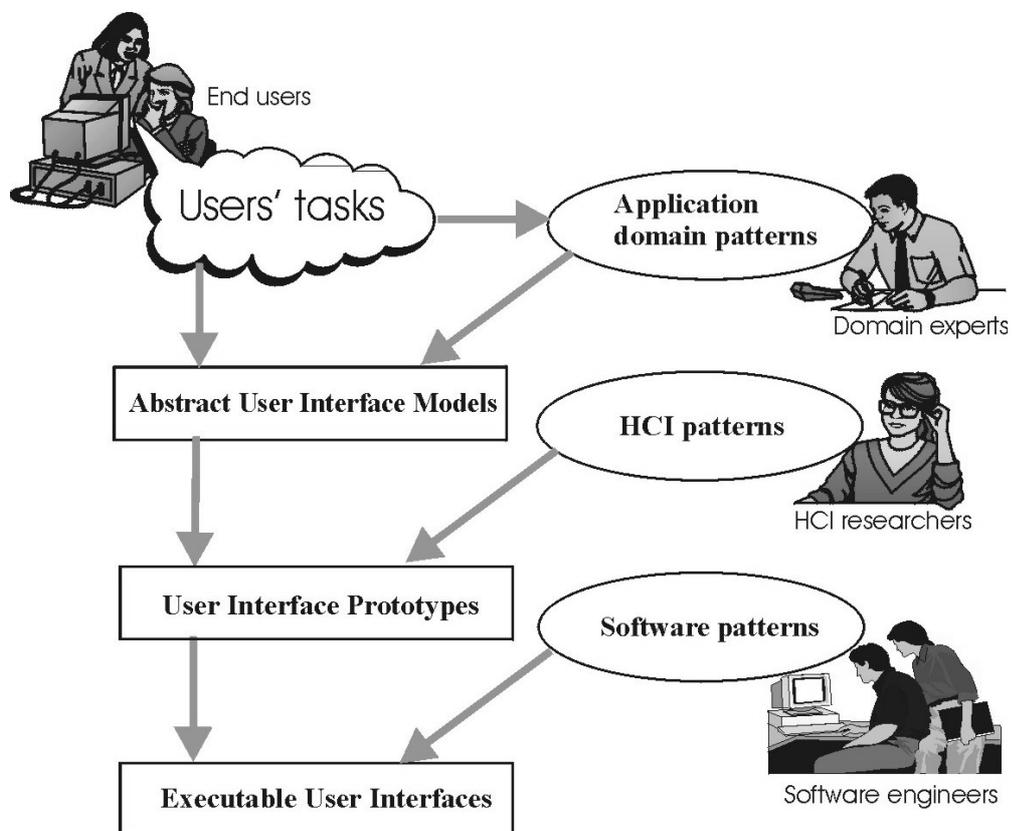
Fortunately, several research works (e.g., [121]) show that the task-based approach which exploits users' task models as the starting point to create user interfaces could be helpful to support user-centered design development process. But these works have neither touched the task modeling methodology nor mentioned the issues such as how to use the well-proven HCI models and software models in a coherent way.

As described in section 3.1.2, design patterns could act as a universal modeling methodology in software engineering, HCI, and application domains. All the related models: users' task models, HCI models, software models could be specified as design patterns: application domain patterns, HCI patterns, and software patterns. These design patterns cut across several disciplines constitute an interdisciplinary pattern system. Research shows that interdisciplinary pattern system can help create usability user interfaces [12] [30] [39] [132].

In order to aid user-centered design development process and creation of task-oriented user interfaces, here, in this thesis, a framework named *Hot User-Centered Design Pattern System* (*Hot-UCDP*) is proposed. It is based on the task-based approach and the interdisciplinary pattern system.

As Fig. 10 shows, *Hot-UCDP* requires that the very first step of creating a task-oriented user interface is to extract the end users' tasks as application domain patterns. It should be done by domain experts since they are the authoritative persons who know the present task fulfillment

ways, the contradictory aspects, and the possible solutions. These domain patterns can help to “*know the users*” and to avoid producing unusable code. Based on the users’ tasks and application domain patterns, the very first abstract user interface models can be outlined. According to these abstract user interface models, user interface designers apply HCI patterns to design user interface prototypes. In terms of these user interface prototypes, software engineers choose related software patterns and tools to create executable user interfaces. Additionally, this framework could also imply several iterations during the establishment of abstract user interface models, user interface prototypes, and executable user interfaces.



**Fig. 10: Hot-UCDP, a framework for user-centered design development process**

Compared to other user-centered approaches (e.g., [121]), *Hot-UCDP*, one of the main results of this work, emphasizes to use well-proven and well-documented interdisciplinary user interface models, i.e., application domain patterns, HCI patterns, and software patterns, in the development process. Therefore, it seems obviously that *Hot-UCDP* could bring at least the following advantages compared to the state-of-the-art:

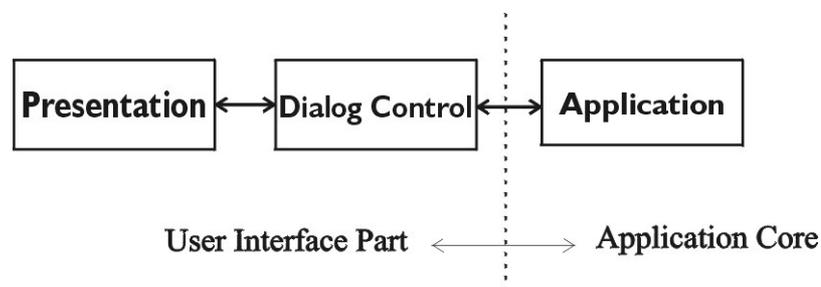
- easier to achieve usability task-oriented user interfaces.
- easier for cooperation of an interdisciplinary user interface development team.
- easier for reuse of all kinds of interdisciplinary models.
- easier for maintenance.

### 3.1.4 Software Architectural Patterns for User Interface Development

*Hot-UCDP* requires to use application domain patterns, HCI patterns, and software patterns to build up task-oriented user interfaces. Application domain patterns and HCI patterns are usually related to a specific user interface, therefore, they will be elaborated through a robot control application in chapter 6 of this thesis. On the contrary, software patterns are usually independent of any specific user interface. They can be used to support large amount of user interface designs. Moreover, software patterns, especially architectural patterns, are very critical to user interface implementation. Therefore, only the related architectural patterns will be investigated here.

#### 3.1.4.1 Seeheim Model

Seeheim model, the very first canonical architecture for user interface development, was developed by user interface researchers at a conference held in Seeheim, Germany, in 1985 [86]. It divides an interactive application into three components: *Presentation*, *Dialog Control*, and *Application*, as depicted in Fig. 11.



**Fig. 11: The Seeheim model**

In Seeheim model, *Presentation* and *Dialog Control* together decide the look and feel of a user interface, where *Presentation* touches directly physical details<sup>12</sup> and *Dialog Control* defines command structures and dialogue sequences. *Application* defines the functionality of the user interface. It contains algorithms and functions specific to application task. All the communication traffic between *Presentation* and *Dialog Control* must go through *Dialog Control*.

However, Seeheim model is just an abstract model. It defines neither concrete communication mechanism nor implementation details for *Presentation*, *Dialog Control*, and *Application*. Moreover, it is not suitable for present object-oriented paradigm since it partitions a user interface merely according to functionality. Therefore, although Seeheim model is the very first architecture model for user interfaces, it is rarely used in present user interface development. Maybe it is the major reason that no one has documented it as a software architectural pattern for creation of user interfaces so far.

### **3.1.4.2 Presentation-Abstraction-Control**

J. Coutaz proposed the Presentation-Abstraction-Control (*PAC*) model in 1987 [24]. The *PAC* model recursively divides an interactive application into a set of cooperating hierarchical related chunks, called *PAC* agents<sup>13</sup>, as shown in Fig. 12. Every *PAC* agent is responsible for a specific functionality of a user interface and is composed of facets<sup>14</sup> of *Presentation*, *Abstraction*, and *Control*. The *Presentation* facet of an agent reflects its visible behavior while the *Abstraction* facet encapsulates its data and functional core. The communication between *Presentation* and *Abstraction* depends on the *Control* facet.

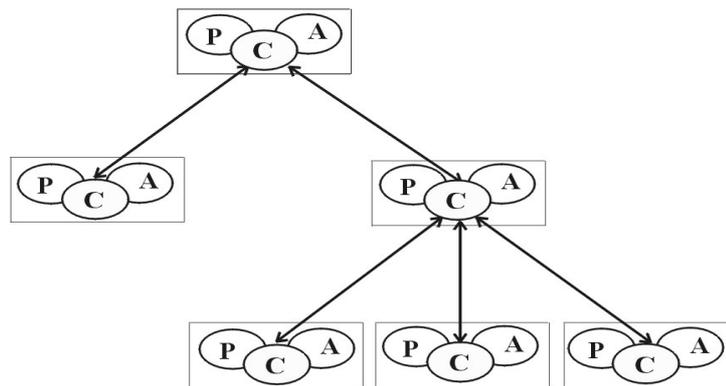
---

<sup>12</sup> Physical details mean the interaction devices and interaction techniques (see Section 2.1.1 of this thesis) of a user interface, which deal with how to capture input from users and how to represent system output to them.

<sup>13</sup> In *PAC*, an agent is a computational unit which has a state, possesses an expertise, and is capable of initiating of and reacting to events [25, p6].

<sup>14</sup> In *PAC*, a facet is synonym of a component.

The top level PAC agent is responsible for providing the whole functionality of a user interface. Its *Presentation* provides the global look and feel aspects while its *Abstraction* includes the global data model. Its *Control* is responsible for communication between its *Presentation* and *Abstraction*, and providing access for its the lower level PAC agents. According to requirement, the top level *Presentation* facet can be further divided into several PAC agents; and their *Presentation* facets can, in turns, be further divided into several PAC agents, recursively.



**Fig. 12: PAC divides a user interface into a set of hierarchical cooperating PAC agents**

Only the bottom level PAC agents deal with the concrete artifacts for HCI. The *Presentation* facet of a bottom level agent presents a view of the corresponding aspects of an interface and provides access to all functions that users can apply to them. Its *Abstraction* facet deals with the data only specific to the agent itself. Its *Control* is responsible for maintaining consistency between its *Abstraction* and *Presentation* and exchanging data with its higher level agents.

Between the top level agent and the bottom level agents, there could be several PAC agents located in the intermediate levels hierarchically. These intermediate level agents are responsible for composition<sup>15</sup> and coordination<sup>16</sup>.

---

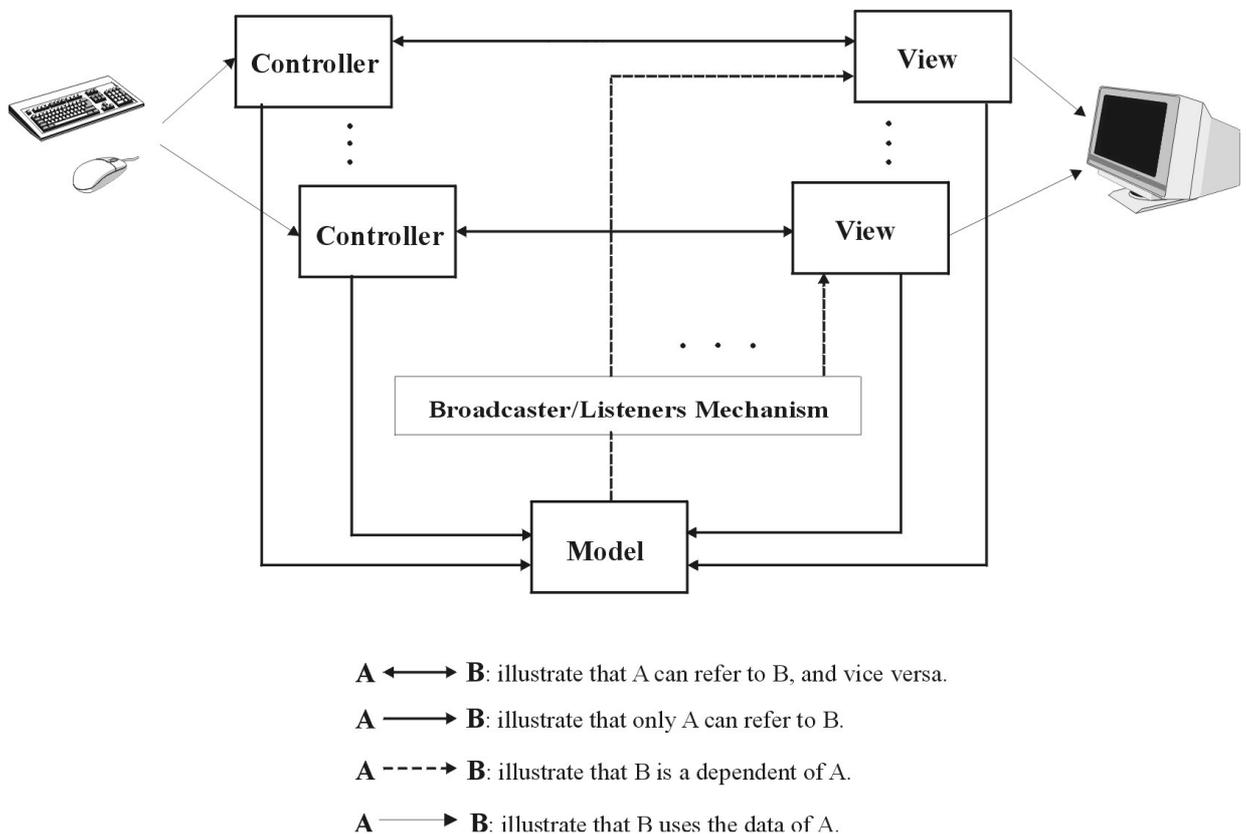
<sup>15</sup> Composition means that a PAC agent groups several lower level PAC agents to form a complex presentation.

<sup>16</sup> Coordination means the maintenance of the consistency among lower level PAC agents.

In [17], the PAC model was documented as the Presentation-Abstraction-Control pattern. With the design pattern modeling methodology, the PAC model is represented in a more readable and reasonable way.

### 3.1.4.3 Model-View-Controller

The Model View Controller (*MVC*) model was firstly used in Smalltalk developed at Xerox PARC in late of 1970s [64]. It is the most widely used model for WIMP user interface design. It and its variants have been built in many systems nowadays, e.g., VisualWorks Smalltalk, Visual Age Smalltalk, Microsoft Document/View Framework, Apple's MacApp Framework, Sun's Java Swing, etc.



**Fig. 13: The MVC Architecture. It splits a user interface into three components: Model, View, and Controller. A Model can have multiple View and Controller pairs and the communication between them depends mainly on the Broadcaster/Listeners mechanism.**

MVC divides a user interface into three components: *Model*, *View*, and *Controller*. *Model* encapsulates data, data structures, and algorithms of an application. *View* presents the system

states and dialog possibility to users. *Controller* is responsible for capturing users' input from the mouse and keyboard; and transfers the input to its *Model* and *View* by message sending.

As Fig. 13 shows, in MVC, one *Model* can have multiple *View* and *Controller* pairs. In addition to message sending, the communication between a *Model* and its *Views* depends mainly on the Broadcaster/Listeners mechanism<sup>17</sup>, where the *Model* is the Broadcaster and the *Views* are the Listeners. According the Broadcaster/Listeners mechanism, once *Model* changes, its *Views* will update themselves automatically.

Note that in MVC only the *View* and *Controller* pairs know the existence of their *Model*, not vice versa. This can improve the flexibility for plugging or withdrawing *View* and *Controller* pairs to a *Model*.

Like PAC, MVC also splits user interface from its application core. Unlike PAC, MVC cuts further the user interface part into two components: *View* and *Controller* pair<sup>18</sup>. The distribution of input from output can reduce the interwoven code and improve the flexibility<sup>19</sup>. Moreover, MVC defines a concrete communication mechanism, the Broadcaster/Listeners mechanism, between *Model* and *View*, which makes the relationship among them more concise.

In [17], MVC has also been documented as a software architectural pattern.

## 3.2 Tools

Generally, a tool is a facility that helps people to carry out a kind of tasks. In addition to sound models, tools are another critical issue for user interface development. In order to make user interface development easier, more than one hundred of tools have been developed [78]. Exhaustively reviewing on these tools can be found in [73] [77].

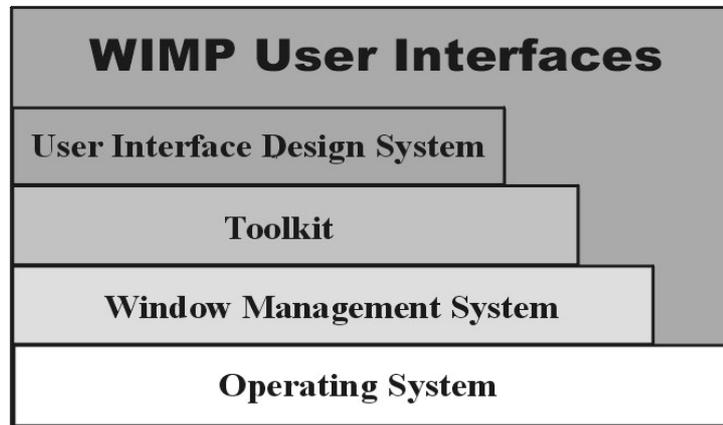
---

<sup>17</sup> The Broadcaster/Listeners mechanism is clearly documented in [36] as the Observer pattern.

<sup>18</sup> In PAC only the Presentation facet can reflect to the input and output of a user interface.

<sup>19</sup> To modify the input (*Controller*) or the output (*View*) is easier since the input is isolated form the output.

Indeed, today, most WIMP user interfaces are developed more or less with the help of tools<sup>20</sup>. Traditionally, window management system, toolkit, and user interface design system are the most popular tools for WIMP user interface development, as shown in Fig. 14.



**Fig. 14: WIMP user interfaces are presently built with diverse tools**

### **3.2.1 Window Management Systems**

A window management system [35] (also called a window manager [73]) is a software package which is built directly on the top of an operating system for managing input (the mouse and keyboard events) from users and output to different areas of a screen (windows) for different contexts. Like the window management system of Window NT 4.0, presently, a window management system is often delivered and interwoven with the system kernel: operating system. Therefore, developing a user interface directly from the window management system level requires high system skill. Furthermore, it does not help to achieve consistent user interfaces, either. Hence, at present few developers build user interfaces directly from this level.

---

<sup>20</sup> In principle, like other parts of an application, a user interface can also be developed from scratch.

### **3.2.2 Toolkits**

Using a window management system as a base, a toolkit is built on a more abstract level. It is a collection of widgets such as menus, buttons, and scroll bars which are mostly provided as classes of an object-oriented programming language<sup>21</sup>. By instantiating the class of a widget according to its well-defined programming interface, developers can create a widget object easier than from scratch. Since different objects of a widget can be created by reusing the same code of a class, a toolkit can guarantee consistent look and feel user interfaces. Because developers can reuse the code of these widgets again and again, toolkits can obviously save the development time.

However, a toolkit provides only programming interfaces. To use it developers must occupy programming skill and master the relevant programming language. For example, to use the toolkit of Microsoft Foundation Class Library, a developer must be proficient at C or C<sup>++</sup>.

### **3.2.3 User Interface Design Systems**

A user interface design system is a high level development tool. In addition to providing widgets to developers, it encompasses other mechanisms for user interface development, too, e.g., the mechanism for creating and managing an application window.

One example of user interface design systems is interface builder. An interface builder often provides graphical tools that help developers create dialog boxes, menus, buttons, and other widgets. For example, the user interface builder of VisualWorks Smalltalk<sup>22</sup> provides a Palette tool showing the widgets available in its toolkit and allows developers to select and position the desired widgets on a Canvas tool with the mouse. By selecting and positioning, a user interface

---

<sup>21</sup> There are toolkits implemented by other kinds of programming languages, too.

<sup>22</sup> The user interface builder of VisualWorks Smalltalk consists of a set of visual tools, e.g., Canvas tool, Properties tool, Palette tool, and Menu Editor.

layout can be generated automatically and intuitively [113] [114]. Because interface builders are easier to learn and use, they are quite welcome by a large number of user interface developers.

However, most present interface builders are limited to building the static part of a user interface: layout. To handle the dynamic properties, e.g., arranging the interconnections among widgets, is still needed to be managed by developers from scratch.

### **3.2.4 Software Frameworks**

A software framework [60] [90] is a tool that provides reusable code and design for creation of a family of applications. It can be seen as a semi-finished application since to develop an application with it, developers just need to add code for the application specific part, most other code and design can be reused from it. It supports reuse at a larger granularity than classes [50].

Although with the tools of window management systems, toolkits, and user interface design systems user interface developers can exploit widgets and arrange interface layouts and contents to rapidly achieve interface prototypes, all these tools are short of support of building dynamic aspects of user interfaces, e.g., interconnections among widgets and application cores are still needed to be built from scratch.

Since a software framework can provide both reusable code and design for building both static and dynamic aspects of an application, it has been widely used in software development at present. And it has been widely used in development of user interface part, too. For example, the application framework of VisualWorks Smalltalk, Visual Age Smalltalk, Microsoft Document/View, Apple's MacApp, Sun's Java Swing, etc., have been widely used to build up user interfaces since the beginning of 1980s.

## **3.3 Pattern-based User Interface Frameworks: an Effective and Efficient Way for User Interface Development**

*Hot-UCDP*, as described in section 3.1.3, is an interdisciplinary pattern-based framework for support of user-centered design development process. A software framework, as described in the above section, can provide reusable code and design for support of the implementation of executable user interfaces. Both frameworks (*Hot-UCDP* and the software framework) are

important to user interface development. Since *Hot-UCDP* has been clearly explored in section 3.1.3, here only pattern-based software framework will be investigated.

As already described in the above section, since software framework can provide well-proven code and design for reuse, it has been widely used in user interface and other parts of a software development. But developing a high quality software framework is hard since a software framework is, actually, an extraction of a set of similar applications. It requires deeply understanding of an application domain and substantial experience of building a set of similar applications for it. On the other hand, to learn and reuse a framework for construction of an application are non-trivial, too [8] [32].

Fortunately, software architectural patterns can provide guidelines to build up quality software frameworks since they record sound system structures from a set of similar applications. Because they are denoted in an intuitive and understandable way, they can help learn and reuse software frameworks, too. Presently, pattern-based software framework is the most promising technology in software community. Many academic and industrial software frameworks have been built with design patterns. For example, SanFrancisco, a software framework for building business applications developed by IBM, is built upon design patterns [54] [116]; ACE, a software framework for building applications with adaptive communication environments, is also built with design patterns [84].

In user interface engineering, to support the implementation phase, research is also going on with pattern-based software framework. For example, as introduced before, the MVC architectural pattern is integrated in several software frameworks: the application framework of VisualWorks Smalltalk, Visual Age Smalltalk, Microsoft Document/View, Apple's MacApp, Sun's Java Swing, etc. These software frameworks can help to ease the burden of developers greatly. For example, the application framework of VisualWorks Smalltalk embraces an intuitive user interface builder and the MVC based framework. With the intuitive Palette tool and Canvas tool, a developer can easily create the layout and content of a user interface by selecting the desired widgets from the Palette and positioning them within the Canvas with the mouse. With the well-established MVC architecture, the layout and content, i.e., the related *View*, can easily be plugged with its *Controller* and *Model*.

### **3.4 Implications for WIMP<sup>+</sup> User Interfaces**

Since, as described in section 2.1.3, Non-WIMP user interfaces should be task-oriented, and since *WIMP<sup>+</sup> user interfaces*, as described in section 2.2.2, are a kind of Non-WIMP user interfaces, *Hot-UCDP*, as introduced in section 3.1.3, which aims to support creation of task-oriented user interfaces, could be used to support developing Non-WIMP and *WIMP<sup>+</sup> user interfaces*. Accordingly, to create a Non-WIMP or *WIMP<sup>+</sup> user interface*, related application domain patterns, HCI patterns, and software patterns are required. In this requirement of *WIMP<sup>+</sup> user interfaces* will be presented in this work. Patterns covering all aspects of Non-WIMP user interfaces need to be developed in future.

On the other hand, since a pattern-based software framework helps create user interfaces effectively and efficiently, it could be important and beneficial to investigate which architectural patterns and software frameworks are adequate to help create *WIMP<sup>+</sup> user interfaces*.

In the previous study, it is clear that PAC and MVC are two widely used software architectural patterns for user interface development so far.

*However, PAC and MVC are just limited to dealing with the characteristics of WIMP user interfaces! They do not provide mechanisms to handle all aspects of WIMP<sup>+</sup> user interfaces. For example, they do not deal with continuous, parallel, real-time, and high bandwidth input and output aspects of WIMP<sup>+</sup> user interfaces.*

Take the most widely used MVC as an example, traditionally, *Controller* of MVC receives input only from the mouse and keyboard, and *View* displays only 2-D graphics or text on a 2-D flat screen. *It does not touch with how to capture data from a sensor and how to drive an effector to leverage a controlled process.*

*Obviously, to support developing WIMP<sup>+</sup> user interfaces, the capability of PAC and MVC needs to be extended!*

For this reason, this doctoral work puts its most effort on researching and documenting a software architectural pattern in order to support developing *WIMP<sup>+</sup> user interfaces*.

### 3.5 Summary

This chapter was intended to provide neither an exhaustive history of the past nor a full scale of future's projection of models and tools for user interface development. It was, rather, to provide a context for *WIMP<sup>+</sup> user interfaces*.

From the view of user interface engineering, user interface models can be divided into two kinds, high level models and low level models. Three representative high level models were surveyed: life cycle model, usability engineering methodology, and the state-of-the-art of user-centered design methodology. Among them user-centered design methodology is the most promising one since it requires an interdisciplinary development team and emphasizes end users' active participation. To aid the individuals of an interdisciplinary user interface development team in clearly expressing their design ideas and communicating, the universal modeling methodology, design patterns, and its applications in software engineering, HCI, and application domains were overviewed. Meanwhile, a task-oriented interdisciplinary pattern-based framework, *Hot-UCDP*, which aims to aid user-centered design development process and the creation of task-oriented user interfaces, was proposed.

Among the low level models required by *Hot-UCDP*, several known software architectural patterns, i.e., Seeheim model, PAC, and MVC, were studied here in this chapter.

Tools for user interface development were also overviewed in this chapter. And it concluded that presently pattern-based software framework is one of the most promising tools for building the dynamic aspects of a user interface.

Through study, it is clear that traditional software architectural patterns and software frameworks can not well support developing *WIMP<sup>+</sup> user interfaces*. For this reason, this work spends its most effort to investigate *WIMP<sup>+</sup> user interface* architectural patterns and frameworks.

In chapter 4, based on the research result of the thesis author, a software architectural pattern for developing of *WIMP<sup>+</sup> user interfaces* will be expounded.

## 4 ACEE: A Software Architectural Pattern for Developing WIMP<sup>+</sup> User Interfaces

### 4.1 About the Pattern Description

As described in section 3.1.2.3.1, software patterns cover various ranges of scale and abstraction. A software pattern can be used to support other larger patterns; and it can be built with other smaller patterns. According to the range of scale, software patterns can be divided into three categories: architectural patterns, element patterns (design patterns), and idioms [17]. Among them architectural pattern is on the highest level. An architectural pattern could be seen as a pattern language to a family of applications since it consists of a set of element patterns which provide a set of components (or subsystems) with appropriate functionalities and includes rules for organizing them together. It is fundamental to an application since it determines the quality criteria such as robustness, flexibility, maintainability, and other system wide structural properties.

In order to fulfill the demands of providing an architecture for *WIMP<sup>+</sup> user interfaces*, in this chapter, a software architectural pattern: *Acquisition-Computation-Expression-Execution (ACEE)*, which was documented by the thesis author and presented at PLoP1999 [128], will be unfolded. It is revised according to the comments and suggestions of improvement obtained from the writer's workshop at PLoP1999 and other resources.

Beside, documentation about a pattern seems to be repetitive and redundant. This style is in common use. Also, it is propagated here.

## 4.2 The ACEE Pattern

### Name

*Acquisition-Computation-Expression-Execution (ACEE)*

### Intent

The *ACEE* architectural pattern aims to help construct the software architecture of *WIMP<sup>+</sup> user interfaces*.

### Context

Traditional WIMP user interfaces help people successfully in office work. But they are short of strategies to deal with continuous, parallel, real-time, high bandwidth input and output which are related to *controlled processes*. *WIMP<sup>+</sup> user interfaces* aim to overcome this pitfall. In a *WIMP<sup>+</sup> user interface*, in addition to WIMP input and output devices like the mouse, keyboard, and 2-D display, *sensors* gather data<sup>23</sup> from a *controlled process* and *effectors* are used to leverage the process according to the control decisions. The control decisions are produced in terms of the input data from the user, *sensors*, and the integrated domain specific control rules. And they have to be produced in the form of control signals recognized by *effectors*.

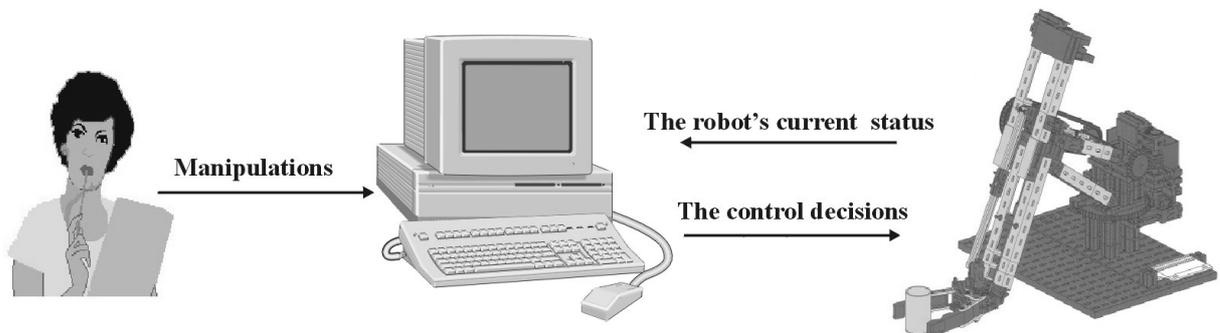
*ACEE* architectural pattern aims to aid in constructing robust, pluggable, flexible, reusable, and maintainable *WIMP<sup>+</sup> user interfaces*.

---

<sup>23</sup> Data here indicates both continuous analog signals and discrete events.

## Example

Suppose there is a *WIMP<sup>+</sup> user interface* for controlling a robot to lift an object to a target position, as shown in Fig. 15. In order to control the robot, the interface should include the code for driving the motors of the robot for moving its arm, opening and closing its gripper. It should integrate the control rules for deciding to where the object should be moved forward and how it will be moved. The control decisions should be regulated timely according to the position of the arm and the width of the gripper. To allow the operator to change the system behaviors (e.g., to change the target position of the robot arm), the interface should express the robot's current status (the present arm position and the width of the gripper) and manipulation possibilities (e.g., to where the robot can be moved) in an intuitive way (e.g., 2-D graphical representation). And the input command from the operator has also to be translated into concrete actions (e.g., the command that is used to move the robot arm to a concrete position has to be translated into a sequence of switching motors on and off). But first the interface has to decide if the command is valid.



**Fig. 15: A WIMP<sup>+</sup> user interface for controlling a robot**

## Problem

What should the architecture of a *WIMP<sup>+</sup> user interface* look like to be highly robust, flexible, pluggable, reusable, maintainable, and efficient to develop?

## Forces

Development of a *WIMP<sup>+</sup> user interface* cuts across several technologies. It does not only deal with computer science but also other engineering techniques. First, it requires that the developers be familiar with software development and necessary hardware technology such as knowing the characteristics of peripheral devices. Second, in order to achieve high usability, knowledge about arranging the layout and content of the user interfaces are required. Third, expertise to application domain is critical, too. All these imply that building a *WIMP<sup>+</sup> user interface* requires an interdisciplinary team. As in the robot example, in order to acquire signals from the robot and to drive the robot moving forward, knowledge about robotics is required, which is generally the purview of electrical engineers. For arranging low level communication between the robot and the computer, software engineers need to be proficient in system programming. To achieve high usability, user interface developers should map the real world control objects and environments into the user interfaces. For making the control decisions, the robot kinematical model as well as the relevant control algorithms are required to be summarized and integrated, which is the special skill of mathematicians. If the user interface architecture is not well constructed, it is hard to adequately assign the development tasks to the individuals within such an interdisciplinary team. This could require that each of them knows the concrete code written by others during the development which could result in low work efficiency. On the other hand, the interwoven code could increase maintenance difficulty, too.

Additionally, the peripheral devices are manifold. The hardware technology changes so fast that in addition to keyboard, mouse, and 2-D display, new and more powerful interaction devices are continuously appearing in the market, which result in a strong demand for the system to flexibly support new devices. Moreover, different operators may need different ways<sup>24</sup> to interact with the system. For example, a small monitor in a factory hall may only offer text output or there may be some standard visualization tool that has to be connected to the system. This implies

---

<sup>24</sup> Sometimes only text is sufficient, some users can deal with only graphical user interface, and sometimes voice information is required.

that the part which deals with peripheral devices and human interaction are prone to change. But, the control rules of a specific application domain are relatively stable. If the software architecture is not well constructed, it is hard to change its parts since to change any part might even be necessary to rewrite the whole application.

Although there are several user interface architectural patterns which deal with some above aspects, e.g., MVC can efficiently and effectively support developing high maintainable, reusable, robust, flexible user interfaces in office environment, unfortunately, none of them deals exhaustively with the innovation features of *WIMP<sup>+</sup> user interfaces*, i.e., the continuous, parallel, real-time, and high bandwidth input and output, in detail.

Shortly, there are three major forces during the development of a *WIMP<sup>+</sup> user interface*:

- The interdisciplinary development team members on the one hand need to closely cooperate with each other, on the other hand each of them needs to concentrate on their own professional work.
- Rebuilding the whole application is hard, but its user interface part is often required to change.
- Present user interface architectural pattern deals with only WIMP interaction techniques and devices, however *WIMP<sup>+</sup> user interfaces* have already gone beyond the WIMP metaphor.

## **Solution**

In order to release the above forces, *ACEE* divides a *WIMP<sup>+</sup> interface* into four components:

- *Acquisition*

Acquires signals from *sensors* and manipulation devices for the status of the *controlled process* (e.g., robot arm position) and user's intention (e.g., control events generated by mouse movements).

- *Computation*

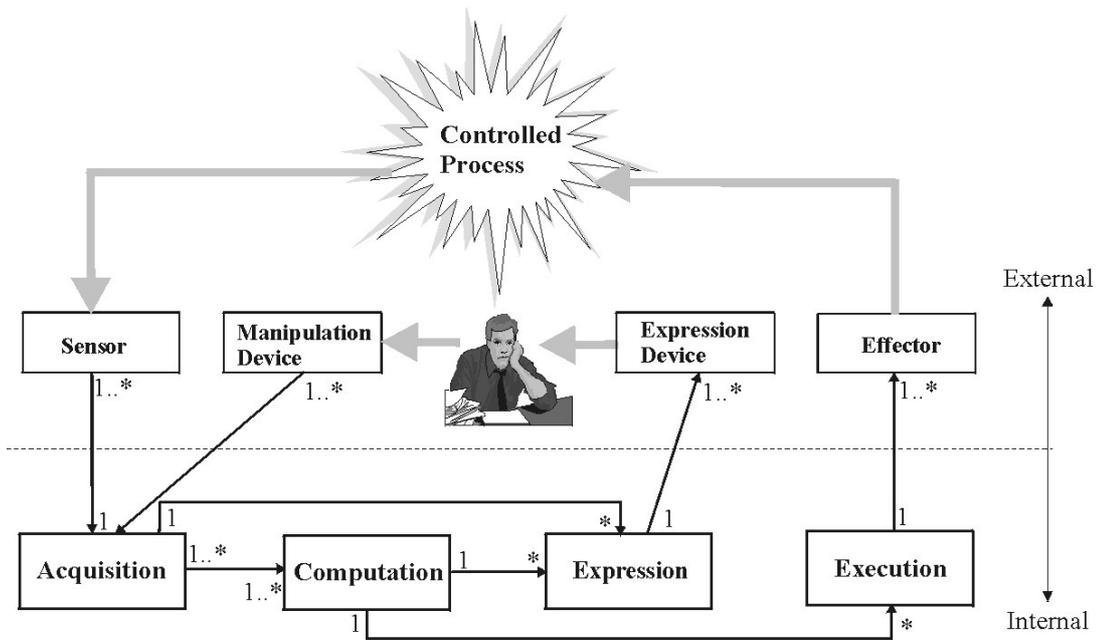
Makes the control decisions. It integrates data model, algorithm, and knowledge base specific to the application domain (e.g., robot kinematical model).

- *Expression*

Represents the acquired data and computation results and provides manipulation possibilities to users. It can be any interaction style what a user can see, hear, or feel. By use of visualization, audio and other possible technology, it expresses the states of the system and provides the manipulation possibilities to users as required.

- *Execution*

Produces commands for driving *effectors* to fulfill the control decisions.



**Fig. 16: The system structure of ACEE**

The principle of ACEE is: once *Acquisition* obtains new data from *sensors* or manipulation devices, it will inform the related *Computation* and *Expression*. *Computation* will make the control decisions upon it and *Expression* will express the data to its user as required. After *Computation* has made the control decisions, *Expression* will express the control decisions to its user in the demanded ways and *Execution* will drive *effectors* to fulfill the control decisions. The major data flow among the components of *Acquisition*, *Computation*,

*Expression*, and *Execution* is unidirectional. That is, it is necessary for *Computation* and *Expression* to know the change (acquired data) in *Acquisition*, and for *Expression* and *Execution* to know the change (control decisions) in *Computation*, but not vice versa. To accomplish this communication relationship, the Broadcaster/Listeners mechanism (also known as the Observer pattern [36]) is exploited. For one *Acquisition* component, developers can build several *Computations* and *Expressions* as its Listeners. And several *Expressions* and *Executions* can be built as Listeners of a *Computation*, too. Due to use of the clearly defined components and the well-established communication mechanism, to integrate or change different acquisition interfaces, control models, expression forms to meet diverse demands is easily achieved.

The system structure of *ACEE* is depicted as Fig. 16. The obligations of each *ACEE* component can shortly be summarized as follows:

- *Acquisition*
  - Monitors the interface connected to *sensors* and manipulation devices for occurring events and signals.
  - Acquires the data coming from the interface.
  - Translates the acquired data to a computer processable format if necessary.
  - Informs all its *Computations* and *Expressions* about its change after new data is acquired and translated.
- *Computation*
  - Keeps its state consistent with that of its *Acquisition*.
  - Makes the control decisions after a change in its *Acquisition*.
  - Informs its *Expressions* and *Executions* about its new state.
- *Expression*
  - Keeps its state consistent with that of its *Acquisition* and/or its *Computation*.

- Translates the expression contents to the data format needed by the expression devices (e.g., large display, acoustic card, etc.) if necessary.
- Expresses the status of the *controlled process* and the control decisions of its *Computation* as requirement.
- Provides the manipulation possibilities to the operator.
- *Execution*
  - Keeps its state consistent with that of its *Computation*.
  - Translates the control decisions from its *Computation* to the data format recognizable by *effectors*.
  - Drives *effectors* to carry out the control decisions.

## Dynamic

Let's explore the dynamic behaviors of *ACEE* through a scenario.

Scenario: in the robot example, the operator uses the mouse to manipulate the representation of the robot gripper on a screen to move the robot arm.

The dynamic behaviors of this scenario can be illustrated as follows:

- *Acquisition* acquires the signals from *sensors* and manipulation devices.

*Acquisition* includes the mechanism for monitoring and capturing data that occurs in the interface of *sensors* and manipulation devices. *Acquisition* then transforms the acquired data to the required format if it is not directly processable by the computer. After that, *Acquisition* notifies its *Computations* and *Expressions* about its change.

In the example, *Acquisition* continually acquires the location of the robot gripper and translates it to the format processable by the system and informs its *Computation* and *Expression*. It also currently captures the manipulation events from the operator, if applicable.

- *Computation* makes the control decisions.

When a *Computation* is notified that new data has been acquired by its *Acquisition*, it will make the control decisions based on the integrated control rules and the acquired data. After the control decisions have been made, *Computation* will notify its *Expressions* and *Executions*.

In the example, *Computation* records the gripper location for future use. Once the manipulation from the operator has arrived, *Computation* will decide if the manipulation is executable according to the integrated rules and the current gripper location. It then informs its *Expression* and *Execution* about its change afterwards.

- *Expression* expresses system information and provides manipulation possibilities to the operator.

When an *Expression* is informed about new data from its *Acquisition* or new control decisions from its *Computation*, it will translate the data to the format required by its expression devices if needed. *Expression* then represents the relevant information and provides manipulation possibilities to the operator in the demanded ways.

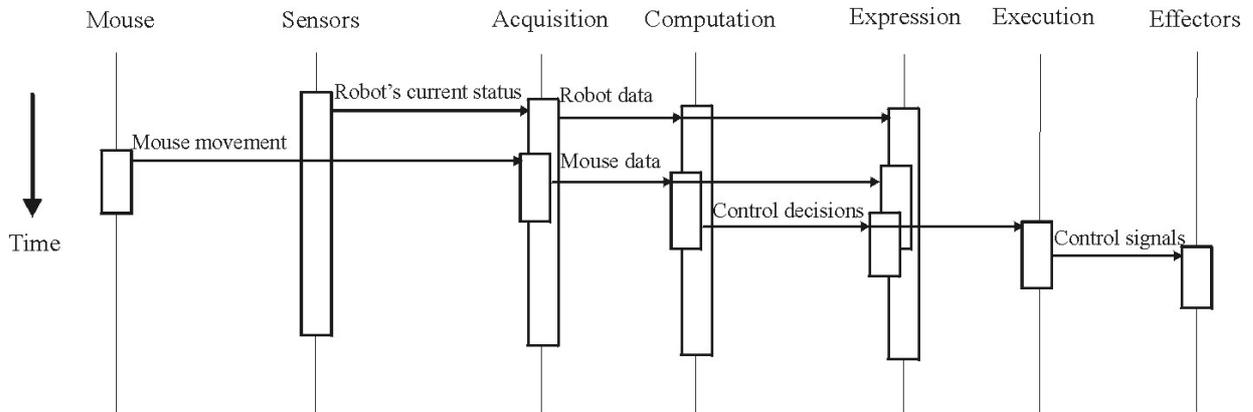
In the example, *Expression* displays the representation of the current location of the robot gripper on the monitor. It also provides manipulation possibilities in terms of the system decisions to the operator, too.

- *Execution* drives *effector* to fulfill the control decisions.

When an *Execution* is notified of new control decisions from its *Computation*, it will transform the control decisions to the signals recognizable by *effectors*. It then drives *effectors* fulfilling the control decisions.

In the example, if manipulation from the operator is confirmed by *Computation*, *Execution* will drive its *effectors*, i.e., the motors, to move the robot gripper forward.

These behaviors can be depicted as an interaction diagram in Fig. 17.



**Fig. 17: The interaction diagram of an ACEE scenario**

## Implementation

The implementation of the *ACEE* architecture consists of eight steps. Step 4~7 each may be repeated several times to define several *Acquisition*, *Computation*, *Expression*, and *Execution* components in order to meet the demands of a *WIMP<sup>+</sup> user interface*.

### Step 1: Construct two Broadcaster/Listeners mechanisms

The communication within *ACEE* depends mainly on two Broadcaster/Listeners mechanisms. The first Broadcaster/Listeners mechanism is located among *Acquisition*, *Computation*, and *Expression* components, where *Acquisition* is the Broadcaster and *Computation* and *Expression* are the Listeners. The second Broadcaster/Listeners mechanism is located among *Computation*, *Expression*, and *Execution* components, where *Computation* is the Broadcaster and *Expression* and *Execution* are the Listeners. According to the work principles of the Broadcaster/Listeners mechanism, if there is any change in the Broadcaster, by sending a changed message<sup>25</sup> to itself, the

<sup>25</sup> In object-oriented programming, invoking a method of an object is called “sending a message” to the object. A message usually contains the method’s name and other necessary parameters.

Broadcaster will automatically trigger the Broadcaster/Listeners mechanism to activate the update methods<sup>26</sup> in its Listeners, which results in their corresponding actions.

Shortly, the Broadcaster/Listeners mechanism requires that the execution of a changed method in a Broadcaster can automatically trigger the execution of the related update methods in its Listeners.

With VisualWorks Smalltalk, the Broadcaster/Listeners mechanism has been well established as the dependency mechanism [114]. The changed, update, and other relevant methods can be automatically inherited from class *Object*<sup>27</sup>. Any object<sup>28</sup> of Smalltalk can act as either a Broadcaster or a Listener of others; or takes the duplication roles of Broadcaster and Listener within different Broadcaster/Listeners mechanisms.

For the C or C<sup>++</sup> implementation, please refer to the Observer pattern [36]. For the Java implementation, please refer to [23].

Step 2: Implement a connection mechanism between computer and the *controlled process*

The connection mechanism is used to establish connection with the peripheral devices. The Acceptor-Connector pattern [92] can be used to build the mechanism. The Acceptor-Connector pattern consists mainly of two components: *Acceptor* and *Connector*. *Acceptor* is used to passively connect with a peripheral device such as a Web server is used to wait for a client connection. On the contrary, *Connector* is used to actively build up connection with a peripheral device. The Acceptor-Connector

---

<sup>26</sup> In object-oriented programming, the term “*method*” is used to refer to a segment of code which has a name, can be individually invoked, and returns a value or finish an action when fulfilled.

<sup>27</sup> The class *Object* is the root class of Smalltalk. This means that all other classes of Smalltalk are its derivatives.

<sup>28</sup> The term “*object*” here is used as the general term in object-oriented programming to refer to a class or an instance.

pattern can decouple the connection establishment, configuration, and initialization of peripheral devices from the concrete processing execution once the services are initialized. For detailed implementation, please refer to [92].

After the abstract classes of *Acceptor* and *Connector* have been built up, their subclasses which integrate the concrete communication protocols should be implemented.

For example, by integration of the TCP protocol, *TCPAcceptor* or *TCPConnector* can be created as subclasses of *Acceptor* or *Connector*.

### Step 3: Implement a *HandlerManager*

To avoid “*Spaghetti code*”, different schemes should be wrapped in different handlers to process different incoming data in *Acquisition* or *Execution*. The responsibility of *HandlerManager* is to allocate the corresponding handlers to the incoming data according to the service requests. And it can be built up with the Reactor pattern or the Proactor pattern [92]. The Reactor pattern is used to dispatch synchronous handlers to process multiple concurrent service requests from the incoming data. The Proactor pattern is used to dispatch handlers to process the multiple concurrent service requests which are triggered by the completion of asynchronous operations. For detailed implementation, please refer to [92].

### Step 4: Implement the *Acquisition* component

- Implement *Acquisition* as a Broadcaster in the first Broadcaster/Listeners mechanism.
- Implement the acquisition interface to be able to sample continuous analog signals and capture discrete events.

*WIMP<sup>+</sup> user interfaces* touch diverse input devices. The acquisition interface should easily establish connection with them. And the code for configuration and initialization of connection should be isolated from the concrete acquisition scheme. For different service requests, *Acquisition* should allocate different schemes

wrapped in different acquisitionHandlers to process them. For standard input devices, their acquisitionHandlers are often delivered with the devices by manufacturers. But for some non-standard products, user interface developers should implement the related acquisitionHandlers themselves.

To build up the interface, developers should follow:

- Instantiate a subclass of *Acceptor* or *Connector*, e.g., *TCPAcceptor* or *TCPConnector*, to establish the low level communication mechanism with the input devices.
- Instantiate *HandlerManager* for dispatching acquisitionHandlers to service requests.
- Implement the sampling and event-driven methods. The sampling and event-driven methods should be built up according to the characteristics of the input devices. If the signals from an input device is continuous, in order to let the computer be able to process these signals, strategy for analog to digital conversion is required. Generally, the Sampling Theorem<sup>29</sup> [95] can be used to digitize the analog signals and it should be integrated in the sampling method.
- Implement acquisitionHandlers. An acquisitionHandler should encapsulate specific scheme for responding to a specific request.

---

<sup>29</sup> The Sampling Theorem states that a continuous analog signal with a finite frequency band, e.g., between 0 and  $f_{\max}$  Hz, can be completely reconstructed from its sample if the sampling frequency is greater than  $2 \times f_{\max}$ . The Sampling Theorem has been widely used in industry and academia to convert analog signals to digital data.

In the example, data comes possibly from the robot interface, keyboard, or the mouse. Different acquisitionHandlers, which are allocated by a handlerManager, are used to respond different service requests.

- Implement the translate method if required. When the acquired data is not processable by the computer, translate is needed.

In the example, the robot arm's horizontal movement is driven by a motor. A pulse switch is used to monitor the motor's movement. Its signals are processed by robotAcquisitionHandler. The result of robotAcquisitionHandler is a number recording the times that the pulse switch has been switched on and off. Therefore, a translate method is needed to interpret the pulse count into a value, telling how many degrees the arm was moved.

- Implement other necessary methods for *Computation* and *Expression* to access its data.

#### Step 5: Implement the *Computation* component

- Implement *Computation* as a Listener of an *Acquisition* within the first Broadcaster/Listeners mechanism and a Broadcaster of *Expressions* and/or *Executions* within the second Broadcaster/Listeners mechanism when necessary.
- Implement the compute method. The compute method should integrate the data model, algorithm, and knowledge base specific to the application domain for making the control decisions. The control decisions are determined by the integrated control rules, the current status of the *controlled process*, and input from user.

In the example, the compute method integrates the robot kinematical model to decide if the manipulation from the operator is valid.

- Implement the necessary methods for its *Execution* and *Expression* to access its core data.

#### Step 6: Implement the *Expression* component

- Implement *Expression* as a Listener of an *Acquisition* with the first Broadcaster/Listeners mechanism or a *Computation* with the second Broadcaster/Listeners mechanism.
- Implement *express* methods. An *express* method integrates the code for expressing the contents in the expression devices. For the standard expression devices like a computer 2-D display, the developers can use the system toolkit. But for other specific expression devices such as an arbitrary audio device or a wall size display, *express* should include specific code to drive them.

Step 7: Implement the *Execution* component

- Implement *Execution* as a Listener of a *Computation* within the second Broadcaster/Listeners mechanism.
- Implement an execution interface in order to drive *effectors* to leverage the *controlled process*.

To drive *effectors* adequately and to avoid “*Spaghetti code*”, *Execution* should involve the mechanism, a *handlerManager*, to allocate *executionHandlers* which wrap different schemes to drive different *effectors* for executing the control decisions. But first *Execution* should establish the low level connection mechanism with its *effectors*.

To build the interface, developers should follow:

- Instantiate a subclass of *Acceptor* or *Connector*, e.g., *TCPAcceptor* or *TCPConnector*, to establish the low level communication mechanism with its *effectors*.
- Instantiate *HandlerManager* in order to dispatch *executionHandlers* to fulfill the control decisions.
- Implement *executionHandlers*. An *executionHandler* encapsulates specific scheme for monitoring and driving an *effector*.

In the example, *Execution* receives the control decisions from its *Computation*. Its *executionHandlers* which are allocated by a *handlerManager* are responsible for producing commands to leverage the *controlled process*. For example, *robotExecutionHandler* is used to produce commands to activate its *effectors*, i.e., the motors of the robot in order to move the robot arm to the target position.

- Implement a *translate* method if necessary. A *translate* method is needed when the data format is not recognizable by *effectors*.

In the example, the *translate* method in *Execution* transforms the target horizontal position of the robot arm calculated by *Computation*, which is a number denoting how many degrees the robot arm should move, to a number that indicates how many times the pulse switch should switch on and off.

Step 8: Initialize the communication relationships among *Acquisition*, *Computation*, *Expression*, and *Execution* quadruples

When the implementation of *Acquisitions*, *Computations*, *Expressions*, and *Executions* are finished, their changed and update methods should be modified in terms of the needs of a *WIMP<sup>+</sup> user interface*.

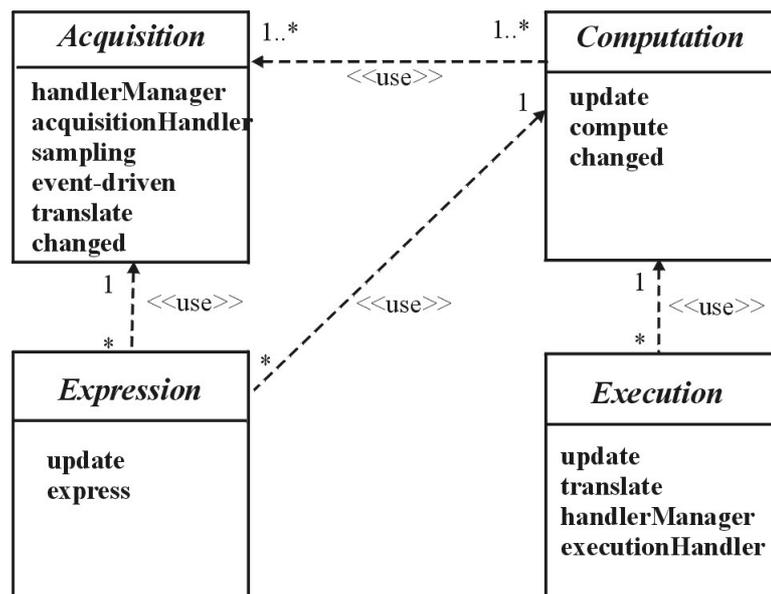


Fig. 18: The abstract class diagram of ACEE

Collectively, the abstract class diagram of *ACEE* can be depicted as in Fig. 18. More detailed class diagram is depicted in Fig. 19.

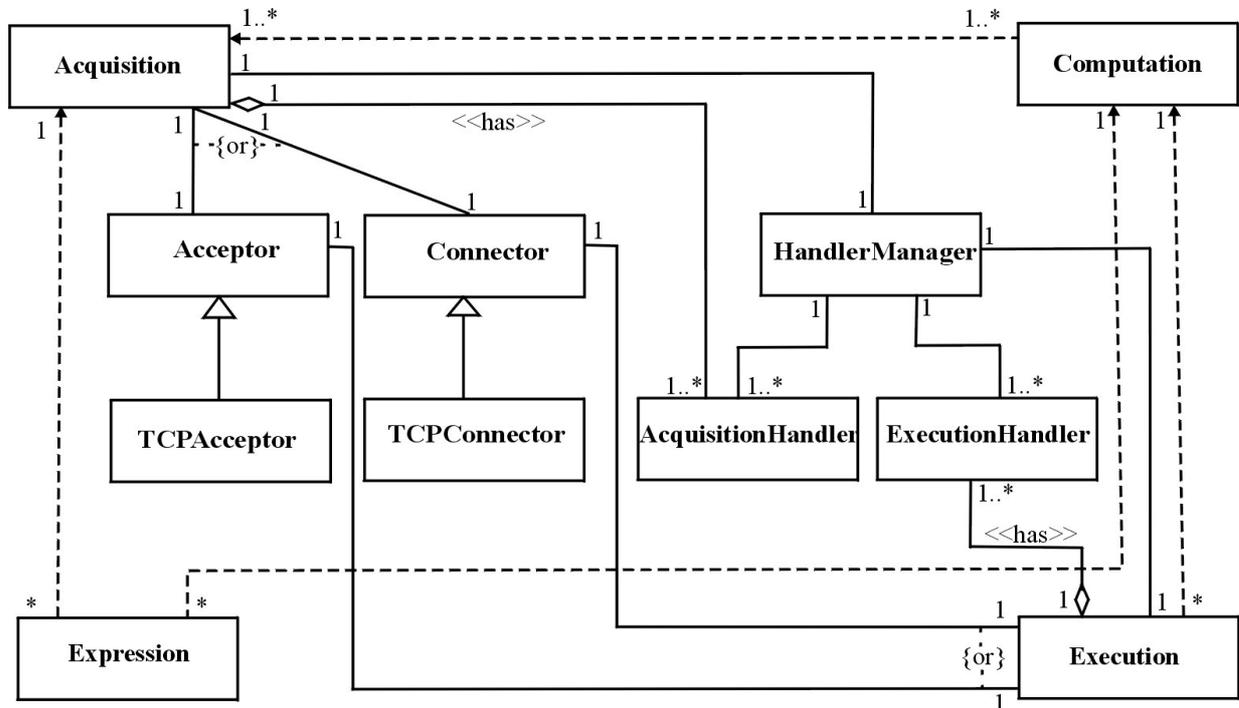


Fig. 19: The detailed ACEE class diagram

### Variant

*ACEE* is a variant of MVC [17], the most well-known architectural pattern for constructing WIMP user interface architecture. It inherits the merits of MVC and extends its ability to meet the demands of building *WIMP<sup>+</sup> user interfaces*.

Exactly, in user interfaces where no *controlled process* is involved, i.e., no *sensors* and *effectors* as input and output, *ACEE* can be simplified to MVC, where *Execution* will disappear, *Acquisition* deals with only the mouse and keyboard events, *Expression* is limited to 2-D graphics or text, and there is only one Broadcaster/Listeners mechanism among *Acquisition*, *Computation*, and *Expression*. Then *Computation*, *Expression*, and *Acquisition* can be seen as *Model*, *View*, and *Controller*, respectively.

## Known Uses

Hot Rolling Mills. A WIMP<sup>+</sup> application for process automation in Hot Rolling Mills developed by Siemens AG [94] covers the whole range between data acquisition via *sensors* (*Acquisition*), computing data in complicated mathematical models (*Computation*), visualizing this data (*Expression*) and controlling the mill with *effectors* (*Execution*). For data acquisition (*Acquisition*), visualization (*Expression*) and low level control tasks (*Execution*), standard products are used that have to communicate with a complex computation component (*Computation*).

SCUT Voice. SCUT Voice is a WIMP<sup>+</sup> application for public telephony voice service developed by the thesis author and her colleagues and students at the South China University of Technology [124]. It was built according to the principle of ACEE. It allows the users to leave, inquire, and delete voice messages through public telephones. It provides one *Voice Mailbox* for each of its users. Each *Voice Mailbox* can hold limited messages. It is divided into four parts: *Acquisition*, *Computation*, *Expression*, and *Execution*. *Acquisition* is responsible for monitoring and acquiring the data from the telephone interface. *Computation* is used to check if the user's operations are illegible and to compute the user's commands. *Expression* displays the system's working states such as which interface channel is occupied and provides dialogue possibility to the system administrator. *Execution* drives the telephone interface for transferring the results, e.g., the voice messages, to the user.

LLDemo. A WIMP<sup>+</sup> *user interface* for robot control [131] developed by the thesis author applies the principle of ACEE to construct the interface architecture, where *Acquisition* is responsible to capture data from *sensors* and manipulation devices, *Computation* decides the system control decisions, *Expression* displays the system's status and provides the manipulation possibility to the operator, and *Execution* drives *effectors*, i.e., the motors, to control the robot move forward. This brings the flexibility of change any component of the user interface without influence of others. Additionally, different components of *Acquisition*, *Computation*, *Expression*, and *Execution* can be developed by different developers concurrently.

## Resulting Context

*ACEE* can effectively support creating *WIMP<sup>+</sup> user interfaces* due to the following advantages:

- *Provides a robust WIMP<sup>+</sup> architecture*

The *ACEE* architectural pattern clearly divides a *WIMP<sup>+</sup> user interface* into four components according to functionality and defines simple but effective communication mechanism among them. This makes each component relatively independent. Therefore, each component can be developed individually without being much interwoven with others which enables user interface developers to concentrate only on the component at hand. Consequently, it could bring robust design.

- *Easily achieves high pluggable, flexible, and reusable design*

Because the Broadcaster/Listeners mechanisms are used, the communication among *Acquisition*, *Computation*, *Expression*, and *Execution* can easily be accomplished. To attach/detach *Computations* and *Expressions* to an *Acquisition* or to attach/detach *Expressions* and *Executions* to a *Computation* is very simple. It can be done even in the running time. This can improve reusability.

- *Improves maintainability*

Since the four *ACEE* components are relatively isolated from each other with respect to functionality, they are easier to understand and maintain.

- *Easily achieves adding, changing, and configuring new input and output devices*

The *ACEE* pattern distills the code for establishing connection, configuration, and initialization of input and output devices from the main part of the user interface by use of the Acceptor-Connector pattern. This makes to add, change, or configure a new input and output device relatively easy.

- *Supports real-time and simultaneous interaction*

A *WIMP<sup>+</sup> user interface* can include several *ACEE* quadruples, each of which is responsible for a thread of interaction. It can include an arbitrary combination of the *ACEE* components, too. For example, several *Acquisitions* can be attached with the same *Computation* to deduce a more complex interaction coordination. This situation often occur in control technology. For example, the control of several robot arms in an industrial assemble application needs to consider the collision constraint. This can be fulfilled by equipping each robot arm with an *Acquisition*. Then all these *Acquisitions* are attached with the same *Computation*, which is used to coordinate the movement of the related robot arms.

- *Supports of capturing both continuous signals and discrete events from sensors and manipulation devices according to their characteristics*

Although many traditional user interface development tools include both polling and event-driven mechanisms for capturing input, they are short of consideration of the characteristics of the input devices in detail (e.g., [114]). This could result in losing important input signals or events. *ACEE* has overcome this drawback by providing both sampling and event-driven mechanisms according to the characteristics of the input devices.

- *Supports synchronous, asynchronous, and distributed operations*

Since some handlers may just do simple data transformations and others may involve time consuming operations, *ACEE* exploits the Reactor pattern and the Proactor pattern to manage the operations of the handlers synchronously and asynchronously. It supports distributed computing, too. For example, according to demands, the *ACEE* components can be mounted in a distributed computing environment.

However, *ACEE* could brings other drawbacks, e.g.,

- *Potentially unnecessary updates in the Listeners*

The Broadcaster/Listener mechanism implies a certain communication overhead which, if not properly taken care of, may slow the system down to an intolerable state. For example, if a Broadcaster has too many Listeners or issues the changed messages too

often, then the consequent updates may take a while to compute. Therefore strategies, such as permitting a Listener to update itself only when an interesting change happens in its Broadcaster, should be applied to limit the potentially unprofitable updates.

## **Related Patterns**

The Broadcaster/Listeners mechanism is used to construct the communication backbone of *ACEE*. For detailed information on it, please refer to the description of the Observer pattern in [36].

The Proactor pattern or the Reactor pattern [92] are used to build *HandlerManager* for dispatching handlers according to the service requests.

The Acceptor-Connector pattern [92] is used for establishing the low level connection with the peripheral devices.

## **Acknowledgements**

Many thanks to Christa Schwanninger who was the shepherd of this pattern and gave lots of concrete advice for its improvement. Many thanks to Prof. Dr. Hans-Jürgen Hoffmann for his fruitful research proposal [50] which guided the thesis author to explore the possibility of extending the MVC pattern for Non-WIMP user interfaces. Special thanks are due to James O. Coplien for his insightful discussion on the pattern writing, Christopher Alexander's idea, and comments on this pattern. Thanks should also go to the attendees of the working group 4 at PLoP1999, Robert Hanmer, Christophe Addinquey, Ku-Yaw Chang, I-ning Chang, Paul Rubel, Jeremy G. Siek, and James O. Coplien (who was the special attendee interesting in this pattern) for their valuable comments and suggestions on this pattern.

## **4.3 Summary**

This chapter expounded the *ACEE* software architectural pattern in detail to meet the demands of dealing with continuous, parallel, real-time, and high bandwidth input and output aspects of *WIMP<sup>+</sup> user interfaces*. A design pattern is not an invention, it is a documentation of a successful solution to a recurring problem. The birth of *ACEE* depends not only on the object-

oriented programming technology of the thesis author but also mainly on her experience in developing applications in communication and control technology closely related to *WIMP<sup>+</sup> user interfaces*.

In addition to sound models, developing *WIMP<sup>+</sup> user interfaces* needs also powerful tools. As described in section 3.3, pattern-based framework is commonly regarded as one of the most widely used and promising tools in both software engineering and user interface engineering. In the next chapter, an *ACEE*-based framework prototype, *Hot-WIMP<sup>+</sup>*, which was developed by the thesis author herself, will be elaborated in detail.