UNIVERSIDADE DA MADEIRA

Object Modeling for User-Centered Development and User Interface Design: The Wisdom Approach

Cê

Duarte Nuno Jardim Nunes

(Licenciado)

Tese Submetida à Universidade da Madeira para a
Obtenção do Grau de Doutor em Engenharia de Sistemas, especialidade de Informática

Funchal – Portugal

April 2001

Supervisor:

Professor Doutor João Bernardo de Sena Esteves Falcão e Cunha
Professor Associado do Dep. de Eng. Mecânica e Gestão Industrial da Faculdade de Engenharia da Universidade do Porto

The research program presented in this dissertation was supported by
the PRODEP Program - 5.2

ABSTRACT

Software engineering evolved significantly in the past decades and is definitively established as an engineering discipline. However, the past years completely changed the nature of the software industry. The Internet, information appliances, entertainment and e-commerce are today large industry segments, often dominated by startups or small companies that work in turbulent environments and face intense competition. Moreover, today there is an increasing diversity of computing devices that are becoming ubiquitous and reaching a growing number of users. In many situations users experience the user-interface of a software intensive system before they commit to purchasing a product or service. Therefore, usability is increasingly becoming an important issue in software development.

This thesis explores how object modeling can contribute to integrate usability engineering into modern software development providing benefits for both disciplines. The Wisdom UML-based approach presented here promotes a lightweight software development method encompassing three major aspects. The Wisdom process enables a new understanding of an UML-based process framework and provides an alternative to the Unified Process specifically adapted to promote a rapid, evolutionary prototyping model, which seamlessly adapts non-classical development lifecycles required by small software development teams. To accomplish this the Wisdom process relies on a new model architecture that promotes a set of UML-based models to support user-centered development and user-interface design. The Wisdom model architecture introduces new models to support user-role modeling, interaction modeling, dialogue modeling and presentation modeling. In addition, we propose a new analysis-level architecture that supports the integration of architectural significant user-interface elements. Finally, the Wisdom notation is a subset and extension of the UML that reduces the total number of concepts required to develop interactive systems. The Wisdom notation simplifies the application of the UML and defines a new set of modeling constructs necessary to support the Wisdom process and architecture.

This thesis also demonstrates how object modeling can benefit usability engineering by leveraging the UML tool support and interoperability. We describe how the Wisdom notation can drive automatic generation of appliance independent user-interfaces and promote artifact change between task modeling tools and UML modeling tools. We also discuss how the Wisdom notation can promote user-interface patterns by providing an abstract notation for documenting recurring solutions in interaction design.

KEYWORDS

Software Engineering

Usability Engineering

User-centered Design

User-Interface Design

Object Modeling

Unified Modeling Language

CASE Tools

RESUMO

A engenharia de software desenvolveu-se significativamente nas últimas décadas afirmando-se hoje definitivamente como uma disciplina da engenharia. Todavia, os últimos anos alteraram por completo a natureza da indústria de software. A Internet, os dispositivos de informação, o entretenimento e o negócio electrónico são hoje mercados importantes, muitas vezes dominados por pequenas empresas que trabalham em ambientes turbulentos e extremamente competitivos. A crescente diversidade e ubiquidade dos sistemas computacionais serve hoje um número crescente de utilizadores que, muito frequentemente, experimentam a interface dos sistemas de software antes mesmo de se comprometerem a adquirir os produtos ou serviços. Desta forma, a "usabilidade" (facilidade de utilização) dos sistemas informáticos tornou-se uma questão central para a engenharia de software.

A presente tese explora a forma como a modelação por objectos pode contribuir para integrar métodos e técnicas da usabilidade no desenvolvimento moderno de software, permitindo benefícios para ambas as disciplinas. O método Wisdom, aqui apresentado, é uma nova proposta de um método de desenvolvimeno de software simplificado que compreende três aspectos importantes. O processo Wisdom define uma alternativa ao Processo Unificado especialmente adaptada ao desenvolvimento de software por pequenos grupos de pessoas de uma forma rápida e assente num modelo controlado de prototipificação evolutiva. O modelo de desenvolvimento subjacente ao processo Wisdom assenta numa nova arquitectura de modelos UML que suporta o desenvolvimento centrado nos utilizadores e o desenho de interfaces com o utilizador. A arquitectura Wisdom introduz novos modelos UML que suportam a modelação de perfis de utilizadores, da interacção, do diálogo homem-máquina, e das questões de apresentação, bem como, propõe um novo enquadramento dos elementos estruturais de análise UML, os quais permitem a integração de aspectos relacionados com a interacção. Finalmente, a notação Wisdom é um subconjunto e extensão do UML que reduz significativamente o número total de conceitos necessários para o desenvolvimento de sistemas interactivos. Por um lado simplifica a utilização do UML e, por outro, define um novo conjunto de elementos de modelação que suportam o processo e a arquitectura Wisdom.

A presente tese demonstra, igualmente, como a modelação por objectos pode beneficiar os métodos e técnicas da usabilidade potenciando o suporte de ferramentas UML e a sua interoperabilidade. Esta dissertação ilustra que a notação Wisdom permite a geração automática de interfaces com o utilizador para múltiplos dispositivos de informação e a troca de informação entre ferramentas UML e ferramentas de modelação de tarefas. É também aqui discutido como a notação Wisdom pode descrever padrões de desenho de interfaces com utilizador recorrendo a uma notação abstracta e normalizada.

PALAVRAS CHAVE

Engenharia de Software

Engenharia da Usabilidade

Desenvolvimento Centrado nos Utilizadores

Desenho de Interfaces com o Utilizador

Modelação com Objectos

Unified Modeling Language

Ferramentas CASE

To my parents who taught me the way

To the memory of my grandmother Cecília who taught me to dare

To Andreia, Tomás and Bernardo that urge me to go further

ACKNOWLEDGEMENTS

I am deeply indebted to many persons who have provided help, support and encouragement. First of all, I would like to thank my supervisor Prof. João Falcão e Cunha for his invaluable help and unselfish support throughout the preparation of this thesis. I thank him for teaching me to be a researcher.

I would also like to thank Prof. José Carmo for his caring and leadership. Warm thanks to Prof. Rita Vasconcelos for believing in my work and making our Department such a pleasant place to work. I would also like to thank all my other colleagues at the Department, in particular Leonel Nóbrega and Jorge Fernandes for their cooperation and fruitful discussions.

Much of the work presented here emerged from the collaboration with the OO&HCI people involved at the different international workshops. Special thanks to Dr. Mark van Harmelen for his vision, commitment and persistence in bringing OO&HCI to the R&D agenda; Dave Roberts for inspiring comments in object modeling for user-interface design; Tom Dayton for raising my interest in participatory techniques and the Bridge; Dr. Srdjan Kovacevic for the fruitful advices on tool issues; Prof. Fabio Paternò for the collaboration on the CTT to UML transformations; John Artim for inspiring comments on Wisdom; Prof. Jaelson Castro for initial collaboration on UCEP; and Prof. Larry Constantine for the availability to share the recent developments on usage-centered design.

I am grateful to all the companies, organizations, practitioners and students who helped me understand their needs and develop Wisdom.

I thank the IBM Ease of Use group, and particularly Roland Merrick, for their collaboration and support in providing the AUIML specs and tools for this project.

Finally I would like to express my warmest gratitude to my family and all my friends for making my life rich and joyful. Warm thanks to Angela and Emílio for their support and for being such great grandparents. I thank my brother Pedro for his friendship and for being such a perfect uncle. I express my heartfelt thanks to my parents for their support, care, and love and also for being such great grandparents - my mother for teaching me honesty and daring, and my father for sharing his life-wisdom and instill that good luck is the product of integrity and hard work.

I give my greatest love to Andreia for sharing her life with me.

Tomás and Bernardo! Daddy is back fulltime – dinosaurs are waiting let's play.

TABLE OF CONTENTS

LIST OF FIGURES

ACRONYMS

ACM – Association for Computing Machinery
API – Application Programming Interface
AUIML – Abstract User-Interface Modeling Language
BPR – Business Process re-engineering
CASE – Computer Aided Software Engineering
CHI – Computer-Human Interaction
CMM – Capability Maturity Model
CSS – Cascading Style Sheets
CTT – ConcurTaskTrees
CUA - Common User Access
CWM - Common Warehouse Model
DHTML – Dynamic Hypertext Markup Language
DMS – Database Management Systems
DTD - Document Type Definition
ERJ – Enterprise Java Beans
GOMS – Goals, Operator, Methods and Selection Rules
GUI – Graphical User Interface
HCI – Human-Computer Interaction
HTML – Hypertext Markup Language
HTTP – Hypertext Transmission Protocol
IDC – International Data Corporation
IT – Information Technology
KLM – Keystroke-Level Model
MBDE – Model-Based Development Environment
MIS – Management Information Systems
MOF – Meta Object Facility
OLE – Object Linking and Embedding
OMG – Object Management Group
OO – Object-Oriented
OO&HCI – Object-Oriented and Human-Computer Interaction
OOA&D – Object-Oriented Analysis and Design

OO-SE – Object-Oriented Software Engineering
OOUI – Object-Oriented User Interface
OOUID – Object-Oriented User Interface Design
OVID – Object, View and Interaction Design
PD – Participatory Design
SE – Software Engineering
SIGCHI – ACM Special Interest Group on Human Computer Interaction
SMIF - Stream-based Model Interchange Format
SPI – Software Process Improvement
SSD – Small Software Development Company
UCD – User Centered Design
UCEP – User-centered Evolutionary Prototyping
UC-OO – User-Centered Object-Oriented
UEL – Usability Engineering Lifecycle
UI – User Interface
UID – User Interface Design
UIDE – User Interface Development Environment
UIML – User-Interface Modeling Language
UIMS – User Interface Management System
UML – Unified Modeling Language
UP – Unified Process
W3C – World Wide Web Consortium
WIMP – Windows Icons Menus and Pointers
WML – Wireless Markup Language
WWW – World Wide Web
XHTML – eXtensible HyperText Markup Language
XMI –XML Metadata Interchange
XML – eXtensible Markup Language
XSLT – eXtensible Stylesheet Language

I. INTRODUCTION

"People Propose, Science Studies, Technology Conforms"
Donald Norman person-centered motto for the 21st century [Norman, 1993]

Software engineering conventionally targeted large, contract-based development for the defense and industry sectors. Although the software engineering discipline achieved significant improvements in that area, today modern software development faces completely different problems. The past years completely changed the nature of the software industry. The increasing diversity of computing devices, the Internet, and entertainment are today large industry segments, often dominated by startups or small companies that work under intense competition. Software engineering is faced today with the problem of adapting conventional methods, techniques and technologies to work in evolutionary, rapid, extreme and other non-classical styles of software development.

There is substantial evidence that attention to usability dramatically decreases the costs and increases productivity of software intensive systems. Nevertheless, developers rarely employ usability engineering methods and techniques in real-life projects. One important reason that usability engineering is not used in practice is the assumption that usability methods and techniques are costly and can only be used by large companies. However, the requirements for the next generation of user interfaces are quite different. The increasing diversity and connectivity of computing devices, and the new consumers of computing technology will have a profound effect on the future of user-interfaces.

Like for software engineering there is no silver bullet to make user-centered development and user interface design easy. Software engineering and usability

engineering are today established fields; decades of research provided significant contributions that can help with the new and envisioned problems. The approach followed in this thesis is to bridge both fields taking advantage of what they best have to offer. The underlying principle is that some of the problems that software engineering faces today are the essence of what usability engineering promoted for years. Conversely, some of the problems that usability engineering currently faces are already successfully solved in the software engineering field.

This thesis explores the bridges between software engineering and usability engineering. Taking one of the most recent and important developments in software engineering – the Unified Modeling Language (UML) – we explore how the benefit of a standard non-proprietary language can help introduce usability engineering methods and techniques into the software development lifecycle.

I.1. MOTIVATION

"*We see computers everywhere but in the productivity statistics*"
Attributed to Robert Solow [Landauer, 1997]

There are several well-documented examples of cost savings from employing usability engineering methods and techniques in software development. Studies consensually point out the evidence that usability engineering dramatically reduces costs (including training, support, errors, maintenance and late changes) and increases the benefits (including productivity and sales). It is not our remit here to make an extensive survey of studies about cost-justifying usability engineering, instead we summarize some references that motivate the need for usability engineering.

In a study about the impact of Information Technology (IT) in labor productivity, Landauer [Landauer, 1997] found out that the low return on investment from IT appears to be the missing piece in the post-war productivity slowdown puzzle. The author demonstrates this hypothesis through an extensive review of productivity statistics that clearly point out that: "the downturn of productivity growth over time coincident with widespread computerization, the concentration of growth failure in industries most heavily involved, the failure of labor productivity in services to respond positively to lavish information technology, the long term lack of correlation of business success with investment in this new technology" [Landauer, 1997]. This interpretation of the productivity statistics is sustained by the fact that computers are: (i) often used to support tasks that are irrelevant or detrimental to true productivity, and (ii) frequently not helpful when intended to increase worker or process efficiency [Landauer, 1997]. According to the author, the reason behind this problem is "foremost is reliance on traditional design and engineering methods that are not sufficient for the development of tools for intellectual work" [Landauer, 1997]. A shift towards user-centered design and usability engineering methods and techniques is predicted by Landauer to increase white-collar work efficiency by 40% per year, against 5% only applying conventional methods [Landauer, 1997].

Landauer's prediction about the impact of user-centered design and usability engineering is sustained by several concrete studies. One study reported savings from usability engineering of $41,700USD in a small application used by 23,000 marketing personnel, and $6,800,000 or a large business application used by 240,000 employees

[Karat, 1990]. Contatine and Lockwood refer an Australian study of hidden usability problems to be between $6,000AD and $20,000AD per year for each workstation [Constantine and Lockwood, 1999]. A mathematical model based on 11 studies suggests that usability engineering saves $39,000USD in a small project, $613,000USD in a medium project, and $8,200,000USD in a large project. In an entire book devoted to the theme of cost-justifying usability [Bias and Mayhew, 1994], Mayhew and Mantei estimate a complete planned usability-engineering program (for a large company in a large project) to cost $132,186USD and the benefits to be $209,490USD for an internal developer organization and $592,635 for a vendor company [Bias and Mayhew, 1994].

The studies mentioned before are usually associated with large companies and large projects. For instance, the cost estimated by Mayhew and Mantei for a usability-engineering program exceeds several times the entire development cost of most small companies. Recognizing the problem that usability engineering is perceived by small companies as expensive (and thus not cost-effective), Nielsen argued that the initial estimate of $132,186USD could be cut to $65,330USD if discount usability engineering methods where used [Nielsen, 1994]. An additional study of the usability budgets of 31 development projects confirmed the assumption that usability costs scale down with project size. Nielsen estimates that two person-years is the mean usability engineering effort in a project and that four person-years would be sufficient for most projects [Nielsen, 1994].

The impact of usability engineering in Internet economy is expected to be of even greater importance. As Nielsen points out "the web reverses the picture. Now, users experience the usability of a site before they have committed to using it and before they have spent any money on potential purchases." [Nielsen, 2000]. Moreover, on the web the competition is not limited to other companies in the same industry: "With all the other millions of sites out there, you are in competition for the user's time and attention, and web users get their expectations from the very best of all these other sites." [Nielsen, 2000].

Finally, usability engineering is also expected to rise in importance with the advent of information appliances and ubiquitous computing. Users are expected to interact with small-specialized information devices that have the ability to share information among themselves [Norman, 1998]. As Norman points out "we're moving from a technology-centered youth to a consumer-centered maturity" [Norman, 1998]. According to the needs-satisfaction curve of technology [Norman, 1998], in the beginning technology cannot meet all the needs of customer and, thus, the early adopters that need the technology are willing to suffer inconvenience and high cost. As technology matures and reaches the point where it satisfies the basic needs customers change their behavior. In the maturity level customers seek efficiency, reliability, low-cost and convenience. Moreover, the new customers that enter the market are more pragmatic, conservative and skeptic [Norman, 1998]. In the words of

Norman: "changing times require changing behavior. The entire product development process must change. The strategy of the industry must change. No longer can sheer engineering suffice. Now, for the first time, not only must the company take marketing seriously, it must introduce yet a third partner to the development process: user experience" [Norman, 1998].

I.2. PROBLEM STATEMENT

The hypothesis of this thesis is that the Unified Modeling Language (UML) can be used to effectively integrate usability engineering into software development with increasing benefits for the development of modern highly interactive software intensive systems. We claim that the same argument that drove the adoption of the UML – to enable tool interoperability at semantic level and provide common language for specifying, visualizing and documenting software intensive systems – applies to bridging the gap between usability engineering and software engineering. In addition, we claim that the problems that software engineering faces with non-classical styles of software development can be solved applying usability engineering methods and techniques.

I.3. SUMMARY OF CONTRIBUTIONS

The main contributions of this dissertation are as follows:

- It describes principles and theories of usability engineering and object-oriented software engineering in terms understandable and relevant to modern interactive system development (see Chapter II);
- It describes the state-of-the-art in object modeling for user-centered development and user interface design. Discusses the useful models for object-oriented user interface design and reviews the integrated user-centered object-oriented methods (see Chapter III);
- It proposes an original process framework that provides an alternative to the Unified Process specifically adapted to promote a rapid, evolutionary prototyping model, which seamlessly adapts the requirements of small software development teams that typically work chaotically (see Chapter IV).
- It proposes an original UML model architecture defining a set of UML models required to build interactive system according to the best practices of object modeling for user-centered development and user-interface design (see Chapter IV). The Wisdom model architecture supports all the artifacts required for user-centered development and user-interface design, in particular user role modeling, interaction modeling, dialogue modeling and presentation modeling;
- It proposes an original extension of the UML analysis framework (the Wisdom user-interface architecture) that enables the integration of architectural significant user-interface elements in the conventional object-oriented analysis framework (see Chapter IV);
- It proposes a set of UML compliant modeling notations (the Wisdom notation), specifically adapted to develop interactive systems (see Chapter IV). The Wisdom notation adapts several recent contributions in the field of OO-UC to the UML style and standard; and also proposes new notations to support effective and efficient user-centered development and user-interface design. The original notational contributions introduced in the Wisdom notation enable the description of user profiles, user-interface architectures, presentation and dialogue aspects of interactive systems;
- It demonstrates how the Wisdom process, architecture and notation collectively support a user-centered object-oriented software development method (the Wisdom method). The Wisdom method is specifically adapted for small teams of developers required to develop high-quality interactive software systems in non-classical environments The Wisdom method is particularly innovative because it combines a controlled evolutionary and rapid prototyping process model (the

Wisdom process), with the best practices defined in usability engineering. Wisdom enables small groups of developers to work in highly turbulent and competitive environments taking advantage of what best characterizes them: enhanced communication, flexibility, fast reaction, improvisation and creativity (see Chapter IV);

- It demonstrates the successful application of the Wisdom method for appliance-independent user interface design, taking advantage of the UML tool interchange infrastructure (see Chapter V). It also demonstrates how the Wisdom models can effectively be used to automatically generate user interfaces rendered on multiple devices;
- It proposes a new envisioned tool environment for appliance independent user-interface design, using flexible model-based automatic generation of user-interfaces capable of being rendered on multiple platforms. It discusses how such an environment could leverage the UML tool interchange format and concentrate on highly focused user-centered tools that help developers on their tasks (see Chapter V);
- It demonstrates how the Wisdom notation can be used to represent the solution underlying user-interface patterns in an abstract way that is independent of a particular design or implementation (Chapter V).

I.4. ORGANIZATION OF THE THESIS

This dissertation consists of six chapters. The next chapter reviews the main theories and practical approaches in usability engineering and object-oriented software development, drawn from the human-computer interaction and software engineering journals and books.

Chapter III describes the recent developments in the field of object modeling and user interface design. The chapter starts with a brief historical perspective and carries on to a discussion of the useful models in object oriented user interface design. The chapter ends with a review of the methods that apply object-oriented principles for user-centered and user-interface design. From the review we present a new proposal for an integrated user-centered object-oriented process framework.

Chapter IV presents the Whitewater Interactive System Development with Object Models (Wisdom) method. The chapter starts discussing the application context of Wisdom (small software companies). The chapter continues presenting the three main contributions underlying the Wisdom method: the Wisdom process, architectural models and notation. Chapter IV ends with a detailed presentation of the Wisdom method for each major software development workflow: requirements, analysis and design. During the presentation specific techniques are discussed and examples provided.

Chapter V describes practical experiences of applying the Wisdom method in different aspects related to user interface design and specifically the impact regarding tool support. In this chapter we demonstrate how Wisdom can be used to automatically generate appliance-independent user interfaces. We then discuss how the Wisdom notation can take advantage of the UML interchange format. The chapter ends with a discussion of the Wisdom notation as a means of representing user-interface patterns.

Chapter VI presents the conclusions and future developments.

II. BACKGROUND: A BRIEF SURVEY OF USABILITY ENGINEERING AND OBJECT MODELING

"The ideal engineer is a composite ... He is not a scientist, he is not a mathematician, he is not a sociologist or a writer; but he may use the knowledge and techniques of any or all of these disciplines in solving engineering problems."
—N. W. Dougherty (1955)

This chapter reviews the main theories and practical approaches in usability engineering and object-oriented software development. The main theories described in this chapter are drawn from the human-computer interaction and software engineering journals and books.

The chapter starts presenting the definition of usability drawn from the user-centered design standard. Usability is then put into the wider context of system acceptability and characterized in terms of a set of attributes (learnability, efficiency, memorability, error prevention and satisfaction) that enable a systematic approach to usability as an engineering discipline. From the definition of usability we then describe the major theoretical frameworks of usability, mainly drawn from the cognitive psychology perspective. From the description of the model human processor framework we present the Goals, Operators, Methods and Selection-rules (GOMS) family of models, one of the most widely accepted theoretical models to predict and evaluate human performance. Although many of these theories are widely accepted, none of them are irrefutable. Therefore we present new developments in the field and describe the major drawbacks of theoretical approaches that describe human behavior using the computer metaphor. One complementary approach, that overcomes the problem of

extracting quantitative measures from a qualitative model, is the notion of mental and conceptual models. A definition of mental and conceptual models is provided and related to Norman's cycle of interaction – another widely used cognitive framework of usability. The cycle of interaction and the gulfs of execution and evaluation, are finally used to review a list of usability heuristics commonly used as principles and rules of user-interface design.

Since the goal of this chapter is to present usability in the context of software development, the remainder surveys recent developments in object modeling as the unifying paradigm of software development. We start from a broad definition of object modeling, as a technique for organizing knowledge, and its application to develop software intensive systems. We then address the nature and importance of modeling in software development, including the major aspects, such as semantic information (semantics), visual presentation (notation) and context. From this broad perspective we shift the focus to object-oriented analysis and design, the field that lead to the Unified Modeling Language *de facto* standard. We present a brief historical perspective of object-oriented analysis and design and use Martin and Odell foundation of object-oriented modeling to introduce the UML standard. Since the UML is a de facto standard, this section is focused on other less known aspects (language architecture and extension mechanisms) that are important for the contributions described in later chapters. The section ends emphasizing the independence of the UML from methodological background and presenting the major process framework of the UML – the Unified Process. Key aspects in this topic are related to architectural descriptions and patterns used in the Unified Process.

The chapter ends with the relationships between usability and software development. In the final sections we briefly describe the most recent and complete proposal for a usability engineering lifecycle. The description includes the tasks required to redesign the software development lifecycle and introduce usability techniques and methods. From this detailed lifecycle we introduce the broader standard for user-centered design, describing the main principles and activities involved. From these two broad process-centric frameworks we briefly describe three areas of usability in software development that had significant developments in the last twenty years. The field of participatory design is presented using a taxonomy that related different techniques that are commonly used in software development, for instance prototyping. The second field concerns conceptual and implementation software architectures for interactive systems that had a major impact during the 80s and early 90s in the implementation of user-interface toolkits. Finally we present a brief overview of model-based design of interactive systems with a special focus in task-based approaches, recognized to be the most influential in the field.

II.1. USABILITY OF SOFTWARE INTENSIVE SYSTEMS: DEFINITION, THEORIES, MODELS AND PRINCIPLES

This section discusses usability of software intensive system, its definition and major theories, models and principles. For the purpose of this thesis we are concerned with usability in the context of the application of engineering principles to systematically build usable and economically viable interactive systems that support real work in an effective and efficient way that promotes satisfaction in use. However, this definition of usability engineering arises from the wider perspective of human-computer interaction (HCI), which is the discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them [ACM, 1992].

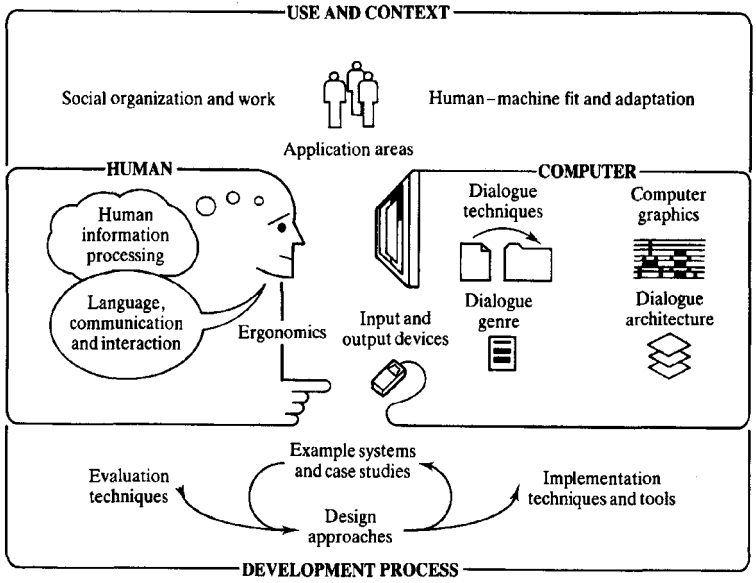


Figure II.1 – The discipline of human-computer interaction (HCI) according to the ACM Special Interest Group in HCI (SIGCHI) [ACM, 1992]

Figure II.1 depicts the ACM SIGCHI vision of HCI. As we can see from this illustration, HCI concerns are much wider than those directly influencing usability engineering. Whenever applicable we use the term usability engineering to stress the focus of this thesis on the theories, models, principles, methods and techniques that are important for systematically building interactive systems.

II.1.1.Definition of Usability

The usability of a product is defined in the ISO 9241 standard, part 11 as:

> *"the extent to which a product can be used by specific users to achieve specific goals with effectiveness, efficiency and satisfaction in a specific context of use". [ISO, 1998]*

This definition relates to the quality of the interaction between the person who uses the product to achieve actual work and the product itself. The important features of the interaction are effectiveness, efficiency and satisfaction. Effectiveness concerns how well the user accomplishes the goals he wants to achieve with the system. Efficiency relates to the resources consumed in order to achieve the goals. Finally, satisfaction concerns how the user feels about the use of the system.

Usability has multiple components and is not a single one-dimension property of a user interface. According to Nielsen [Nielsen, 1993] usability is traditionally associated with the following attributes:

- Learnability – the system should be easy to learn, enabling even inexperienced users to perform rapidly the supported tasks;
- Efficiency – the system should be efficient in use, so that once the user has learned the system he should be able to achieve a high level of productivity;
- Memorability – the system should be easy to remember, allowing casual users to reuse the system without having to learn the system again;
- Error prevention – the system should prevent users from making errors, in particular, errors that damage users work must not occur. The system should enable users to recover from errors;
- Satisfaction – The system should be pleasant to use, fostering subjective satisfaction in use.

The precise definition of such usability attributes enables a systematic approach to usability as an engineering discipline. The components described above can be improved, measured and evaluated, and hence, constitute an alternative to speculative approaches to usability.

Usability is typically measured by testing a number of representative users performing a predetermined set of tasks. To determine the system overall usability we can take a mean value of the scores of a set of usability measures, or, recognizing that users are different, considering the entire distribution of usability measures [Nielsen, 1993]. Furthermore a number of methods for analyzing user interfaces quantitatively are also available, such as GOMS (and its variations) [Card et al., 1983], Hick's law, Fitt's law and Raskin's measure of efficiency [Raskin, 2000] (see section II.1.2.2).

Constantine and Lockwood also provide a set of five rules of usability that point out the general direction towards usability. They provide a general framework for the effectiveness of modern user interfaces in the form of the following usability rules [Constantine and Lockwood, 1999]:

- Access rule – the system should be usable without help, prior experience or instruction by an experienced user in the application domain;
- Efficacy rule – the system should not interfere or impede efficient use by an experienced and skilled user;
- Progression rule – the system should accommodate and facilitate continuous advancement in knowledge, skill and facility as the user gains experience;
- Support rule – the system should support real work by making it simpler, faster, easier and more fun for the users to perform the tasks they are trying to accomplish and by creating new possibilities;
- Context rule – the system should be suited to the operational context (real conditions and environment) within which it will be deployed.

II.1.1.1.Usability and other Considerations

In the words of Nielsen: "Usability is a narrow concept compared to the larger issue of system acceptability" [Nielsen, 1993]. System acceptability concerns the wider extend to which a system satisfies all the needs and requirements of the user and other potential stakeholders directly or indirectly influenced by the system. Acceptability is a combination of social and practical attributes. Given that a system is socially acceptable – meaning that it conforms to the existing social, cultural and moral guidelines – we can enumerate a number of practical acceptability attributes. One example of such attributes is illustrated in Figure II.2.

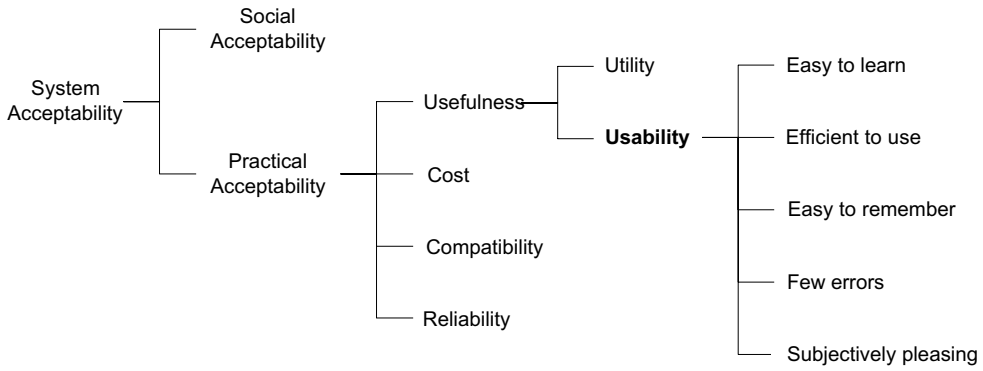


Figure II.2– Nielsen's model of attributes of system acceptability [Nielsen, 1993]

II.1.2.Cognitive Frameworks of Usability

The dominant frameworks used in HCI to understand and represent how humans interact with computers are based on the cognitive psychology perspective. Cognitive psychology is a theoretical perspective that focuses on how human beings achieve their goals in terms of cognitive tasks that involve transforming and processing information from the sensory input.

The models and theories summarized in the following sections are grounded in the assumption - from the cognitive psychology perspective - that humans can be

characterized as information processors. This approach resembles the ones used to describe computer systems in terms of memories, processors, their parameters and interconnections. This description is approximate in the sense that it only intends to predict human-computer interaction and not to explain the physiological processes that happen in the brain.

II.1.2.1.The Model Human Processor

The Model Human Processor is an extension of the basic information-processing model that sees the human mind as a series of ordered processing stages [Preece, 1995]. This simplified model starts with the sensory input stimuli and comprises a four-stage model of encoding, comparison, response selection, response execution, and finally ends with the output.

The two main extensions of the basic information-processing model are the inclusion of attention and memory processes. They form the basis of the Model Human Processor (see Figure II.3) which consists of three interacting systems: the perceptual system, the motor system and the cognitive system – each with it's own memory and processor [Card et al., 1983].

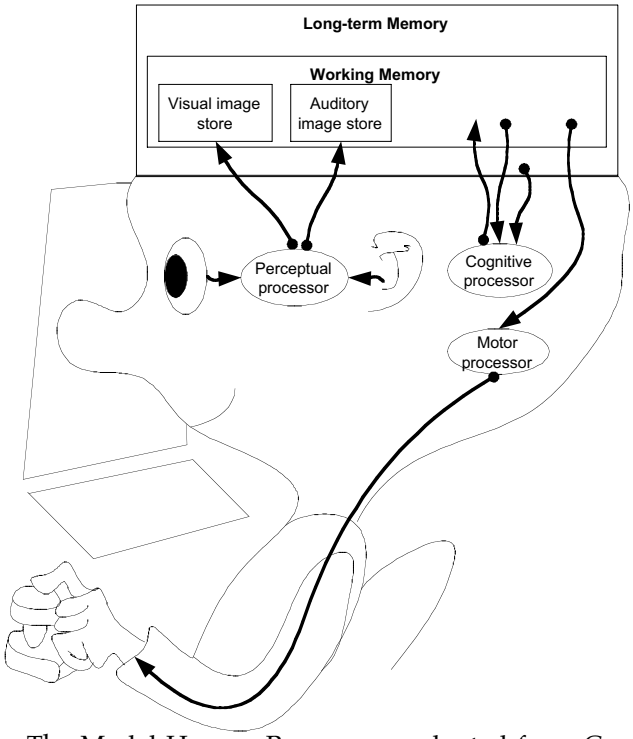


Figure II.3 – The Model Human Processor – adapted from Card et al [Card et al., 1983]

The perceptual system carries sensations from the physical world through sensors (eyes, ears, etc.) and buffers that information while its being symbolically coded. The most important buffers are depicted in Figure II.3 as the visual and auditory stores.

The cognitive system connects the inputs from the perceptual system to the right outputs of the motor system. The process consists in receiving symbolic coded information from the working memory, eventually combining that information with previously coded information in the long-term memory, and decides how to respond. Since most of the human tasks are complex and involve learning, retrieval of facts, or problem solving; the memories for the cognitive system are more complicated. The working memory holds the intermediate products of thinking and the symbolic representations produced by the perceptual system. The working memory consists of a subset of the symbolic elements (chunks) in the long-term memory that are active to support the mental operations. The most important characteristics of the working memory are its limited capacity to hold information, in amount and time. The long-term memory holds the available mass of knowledge in the form of a network of related chunks accessed associatively from the contents of the working memory. The contents of the long-term memory comprise facts, procedures and history. Information in the long-term memory must be encoded in symbolic form – there is no direct way of storing information from the sensory inputs into the long-term memory.

The motor system is responsible for carrying out responses to the sensory inputs. Thought is transferred into action, using information in the working memory, by activating patterns of voluntary muscles.

The Model Human Processor as a Framework for Estimating and Evaluating User Interfaces

The Model Human Processor framework, presented in Figure II.3 and briefly describe before, provides a means of characterizing the various cognitive processes that are assumed to underlie the performance of a task. From this conceptual model several approaches were developed to translate qualitative descriptions into qualitative measures – they are known as GOMS (goals, operations, methods and selection rules) family of models [Card et al., 1983].

The GOMS family of models is based on a set of parameters that describe memories and processors. The most important parameters for memories are [Card et al., 1983]:

- μ - The storage capacity, in items;
- δ - The decay time of an item, and
- κ - The main code type (physical, acoustic, visual, semantic, etc.).

Conversely the most important parameter for processors is:

- τ - The cycle time.

The parameters describing memories and processors can be experimentally estimated according to the physiological characteristics of human beings. Figure II.4 summarizes different estimates for the perceptual, cognitive and motor systems.

Parameter	Perceptual System	Cognitive System	Motor System
Memories			
μ	μ_{VIS} = 17(7~17) letters μ_{AIS} = 5(4.4~6.2) letters	μ_{WM}(pure) = 3(2.5~4.1) chunks μ_{WM}(effective) = 7(5~9) chunks	
δ	δ_{VIS} = 200(90~1000) msec. δ_{AIS} = 1500(900~3500) msec.	δ_{WM}(1 chunk) = 73(73~226) sec. δ_{WM}(3 chunks) = 7(5~34) sec. $\delta_{LTM} = \infty$	
κ	κ_{VIS} = physical κ_{AIS} = physical	κ_{WM} = acoustic or visual κ_{LTM} = semantic	
Processors			
τ	τ_P = 100(50~200) msec.	τ_C = 230(70~700) msec.	τ_M = 70(30~100) msec.

Figure II.4– Typical Values for Parameters Describing Memories and Processors in the Model Human Processor (source: [Card et al., 1983])

Furthermore, the parameters obey a set of principles, called the principles of operation, which enable qualitative predictions and measurements of user behavior. The principles of operation for the Model Human Processor are summarized in Figure II.5.

The Model Human Processor Principles of Operation

- P0. Recognize-act Cycle of the Cognitive Processor – on each cycle of the cognitive processor, the contents of the working memory initiate actions associatively linked to them in long-term memory, these actions in turn modify the contents of the working memory.
- P1. Variable Perceptual Processor Rate Principle – the perceptual processor cycle time τ_P varies inversely with stimulus intensity.
- P2. Encoding Specificity Principle – specific encoding operations performed on what is perceived determine what us stored, and what is stored determines what retrieval cues are effective in providing access to what is stored.
- P3. Discrimination Principle – the difficult of memory retrieval is determined by the candidates that exist in memory, relative to the retrieval cues.
- P4. Variable Cognitive Processor Rate Principle – The cognitive processor cycle time τ_c is shorter when greater effort is induced by increased task demands or information loads, it also diminishes with practice.
- P5. Fitt's law – the time T_{pos} to move the hand to a target of size S which lies a distance D away is given by: $T_{pos} = I_M \log_2(D/S + 0.5)$, where $I_M = 100[70 \sim 120] msec/bit$.
- P6. Power Law of Practice – the time T_n to perform a task on the n^{th} trial follows a power law: $T_n = T_1 n^{-\alpha}$, where $\alpha = 0.4[0.2 \sim 0.6]$.
- P7. Uncertainty Principle – decision time T increases with uncertainty about the judgment or decision to be made: $T=I_C H$, where H is the information-theoretic entropy of the decision and I_C=150(0~157) msec/bit. For n equally probable alternatives (called Hick's Law) $H = \log_2(n+1)$, for n alternatives with different probabilities, p_i, of occurrence, $H = \sum_i p_i \log_2(1/p_i + 1)$.
- P8. Rationality Principle – a person acts so as to attain his goals through rational actions, given the structure of the task and his inputs of information and bounded by limitations on his knowledge and processing ability: Goals + Task + Operators + Inputs + Knowledge + Process-limits → Behavior.
- P9. Problem Space Principle – the rational activity in which people engage to solve a problem can be described in terms of: (i) a set of states of knowledge, (ii) operators for changing one state into another, (iii) constraints on applying operators, and (iv) control knowledge for deciding which operator to apply next.

Figure II.5 – The Model Human Processor Principles of Operation [Card et al., 1983]

II.1.2.2.The GOMS Family of Models

Since Card and colleagues presented the original concept of GOMS, several authors extended the original method [John and Kieras, 1996a; Kieras, 1988; John, 1990; John and Gray, 1995]. They also outlined several significant gaps in cognitive theory to address some HCI concerns - for instance fatigue, user-acceptance and fit to organizational life [John and Kieras, 1996a]. Despite that, the GOMS model is one of the few widely known HCI conceptual theories and it has been widely discussed, experimented and tested.

GOMS is a method for describing a task and the user's knowledge of how to perform the task in terms of Goals, Operators, Methods and Selection rules [John, 1995]. Since the concepts underlying the GOMS model are widely used in the HCI field – sometimes with different meanings – it is useful to conceptualize those terms. Here we adapt the definitions from the three level structure proposed by Preece et al [Preece, 1995] (see Figure II.6).

Concept	Definiton	Example	GOMS
Goal	The sate of the system the human whishes to achieve	Write a letter	**Goal**
Device	*Instrument, method, agent, tool, technique, skill...*	*Mouse, keyboard, direct-manipulation, ...*	
Task	The set of activities required, used or believed to be necessary to achieve a goal.	Edit text, print letter, ...	**(sub)Goal**
Action	An internal task that involves no problem solving or control structure component.	Enter text, move cursor, ...	**Operators, methods**

Figure II.6 – Definitions for a three level Goal-Task-Action Framework

A goal can be defined as the state of the system that the human whishes to achieve. A goal, sometimes called an external task, can be achieved using some instrument, method, agent, tool, technique, skill, or generally, some device that enables the human to change the system to the desired state. A task (or more specifically an internal task) is defined as the activities required, used or believed to be necessary to achieve a goal using a particular device. A task is, hence, a structured set of sequential activities that a human has to do (or thinks he as to do) in order to accomplish a goal. Finally an action is defined as an internal task that involves no problem solving or control structure component.

In GOMS, goals are what the humans want to accomplish with the software, since they involve problem solving and control structure, they relate to goals and tasks in the conceptual hierarchy depicted in Figure II.6. Conversely operators are the actions that the software allows the user to perform. Operators are actions in the conceptual hierarchy of Figure II.6 and they usually map to typed commands, menu selection, button presses, gestures or spoken commands in concrete user interfaces. Methods are

well-learned sequences of sub-goals and operators that can accomplish a goal. A classic example of a method is deleting a paragraph in a word processor – place the cursor at the beginning of the paragraph, drag to the end of the paragraph, release highlighting the paragraph and press the delete key. Finally, if there is more than one method to accomplish the same goal, selection rules apply, that is, decide what method to use in a particular circumstance [John, 1995].

There are at least four different versions of the original GOMS method that achieved a broad use [John, 1995]. The original formulation proposed by Card et al [Card et al., 1983], which is a loosely defined demonstration of how to express a goal hierarchy, methods and operators, and to formulate selection rules. A simplified version, also defined by Card and colleagues, that uses only keystroke level operators and no goals, methods and selection rules. The simplified version is called the Keystroke-Level Model (KLM) and enables simple calculations through lists of keystrokes and mouse movements, supported with a set of simple heuristics used to "place mental operators". A more rigorous version, called NGOMSL and proposed by Kieras [Kieras, 1988], presents a procedure for identifying all GOMS components, expressed in a form similar to an ordinary programming language. NGOMSL includes rules-of-thumb about how many steps can be in a method, how goals are set and terminated, and what information needs to be remembered by the user while doing a task [John and Kieras, 1996a]. Finally John [John, 1990; John and Gray, 1995] proposed a parallel activity version, called CPM-GOMS, which uses cognitive, perceptual and motor operators in a PERT chart to show how activities can be performed in parallel.

Different versions of the GOMS model produce quantitative and qualitative predictions of how people will use a proposed system. The different types of applications of GOMS are skilled performance time, method-learning time, and the likelihood of memory errors. Therefore GOMS can be effectively used to complement other system cost-estimation methods, enabling quantitative estimation of human related costs (costs to train people, costs operating the systems, and costs from possible errors and error recovery) [John, 1995]. However, GOMS analysis is limited to situations in which users will be expected to exhibit skilled performance, that is, it doesn't contemplate problem solving or hunting around – very common in creative or non-repetitive tasks [John, 1995]. Figure II.5 reviews different uses of different versions of the GOMS method.

Version of GOMS	Type of System	Predictions	Uses of Analysis	Task Domain
Original	Single-user passive	Operator, sequence. Performance time. Working memory errors. Learning time.	Method verification. Comparison of systems. Documentation design.	Text editor. Operating system. VLSI CAD. Spreadsheet. Hypercard. Videogames.
Keystroke-Level Model (KLM)	Single-user passive	Performance time.	Method verification. Redesign justification.	Text editor. Spreadsheet. Map digitizer.
NGOMSL	Single-user passive	Operator sequence. Performance time. Learning time. Working memory errors.	Method explanation. Method verification. Redesign.	Text editor. Ergonomic CAD.
CPM-GOMS	Single-user passive, surrogate user	Performance time.	Comparison of systems. Method verification.	Telephone operator workstation. Airline schedule.

Figure II.7– Review of GOMS Family of Methods (adapted from [John, 1995])

II.1.2.3.Recent Developments in Cognitive Psychology Frameworks

Theoretical approaches that describe the activity of the brain using the computing metaphor, and its' underlying concepts (buffers, memories, processors, etc.), were popular and appealing until the 80s. The popularity of these approaches was related to the possibility of testing those models. Since the 80s different approaches have evolved in the HCI field. The new approaches are commonly known as the connectionist and computational approaches [Preece, 1995].

The computational approaches continue to adopt the computer metaphor as the main theoretical framework. However, the emphasis has shifted from the information-processing framework (where the information is processed) to modeling human performance in terms of when information is processed.

The connectionist approaches adopt the brain metaphor in which cognition is represented as a set of interconnected nodes in the form of a neural network or parallel distributed processing framework. Hence, cognitive processes are viewed as activations of nodes in the network and the connections between them instead of the process of manipulating and processing information.

The increasing importance of modeling and analyzing the way people work and interact in the real world, leveraged yet another emerging theoretical framework known as distributed cognition. Distributed cognition aims at providing an explanation of how cognitive activities are embodied and situated in the work context in which they occur. This involves describing cognition as a set of distributed people, computer systems and other cognitive artifacts; and the way they connect to each other in the environmental setting in which they are situated. Distributed cognition has been used mainly in safety critical systems, for instance, air traffic control systems and ship navigation [Hollan et al., 2000].

II.1.2.4.Mental and Conceptual Models

One of the problems of extracting a quantitative model, such as GOMS, from a qualitative description of user performance is ensuring that the two are connected. In particular, Barnard [Barnard, 1987], noted that the form and content of the GOMS family of models is relatively unconnected with the Model Human Processor. One way of dealing with such oversimplified conceptualizations of human-behavior concerns conceptual or mental models.

Mental models refer to representations people construct in their minds of themselves, others, objects and the environment to help them know what to do in current and future situations [Preece, 1995]. When such models refer to software or physical systems they are usually known as conceptual models. A good conceptual model allows humans to predict the effects of their actions: "without a good (conceptual) model we operate by rote, blindly; we do operations as we were told to do them; we can't fully appreciate why, what effects to expect, or what to do if things go wrong (…)" [Norman, 1988].

Norman defines a conceptual user model, in the context of HCI, as "the model people have of themselves, others, the environment, and the things with which they interact. People form mental models through experience, training and instruction" [Norman, 1988].

There are different mental models with respect to the development of interactive systems (see Figure II.8). The design model is the designers' mental model of the system, it reflects the model that designers' build when developing the system. The user model is the mental model humans develop when interacting with the system, through experience, training and instruction. Finally, the system image is the physical (perceivable) structure of the actual system built. Since all communication of the users with the system happens through the system image, if the system image doesn't reflect clearly and consistently the design model there is a good change that users will form wrong mental models.

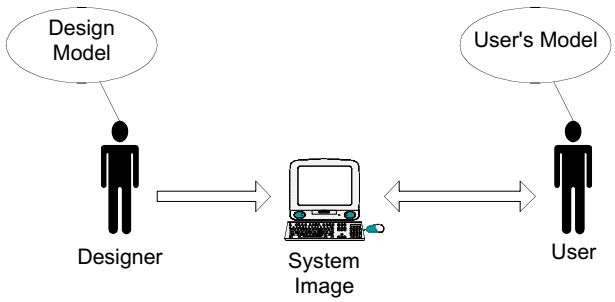


Figure II.8 – Conceptual Models

Different studies discussed whether people actually have and use mental models when interacting with devices and systems [Preece, 1995]. Although this is in sharp contrast with the prescriptive advice in HCI, the utility of conceptualizing the users'

knowledge can be very productive helping designers construct an appropriate model of the system. Furthermore, conceptual models can provide a good heuristic tool. Examples of such utility are evident, for instance, in several usability heuristics described in the following sections.

II.1.2.5.Norman's Cycle of Interaction

Norman's Cycle of Interaction is another well-known cognitive framework of HCI [Norman and Draper, 1986]. This model provides a way of identifying the main phases in a user interaction - the seven stages of action. The author claims that, since the stages are not discrete entities, the cycle forms an approximate model and not a complete psychology theory. This model provides a structured framework for design and evaluation of interactive systems. The seven stages of action are:

- Forming the goal
- Forming the intention
- Specifying an action
- Executing the action
- Perceiving the state of the world
- Interpreting the state of the world
- Evaluating the outcome

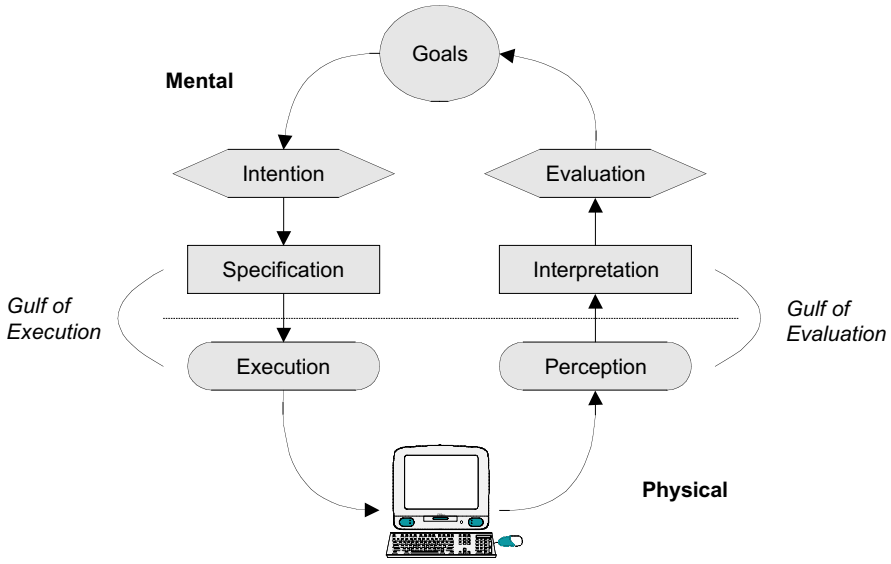


Figure II.9 – Norman's Cycle of Interaction: The Seven Stages of Action and the Gulfs of Execution and Evaluation (adapted from [Norman and Draper, 1986])

Norman's Cycle of Interaction involves two sets of stages related to a common goal - the state of the system we want to achieve (section II.1.2.2). The stages of execution (at the left-hand side of Figure II.9) translate the initial goal into an intention to do something. The intention is then translated into an action sequence (a set of actions

that can be performed to satisfy the intention) that is still a mental event. Finally the actions are executed, that is, performed upon the physical world.

The stages of evaluation (at the right-hand side of Figure II.9) start with the perception of the world. This perception of the physical world is interpreted according to the expectations and then evaluated, that is, compared with respect to the intentions and goals (part of the stages of execution). The Cycle of Interaction corresponds to the coupling of both sets of stages with the common goal.

The Gulfs of Execution and Evaluation

The model reflected by Norman's cycle of iteration enables the identification of several gulfs that separate mental intentions and interpretations from physical actions and states. Each gulf indicates a potential problem for the users since it reflects the distance between the mental representations of the users (conceptual model) and the physical components and states of the environment (physical model) [Norman and Draper, 1986].

- The gulf of execution – "The gulf of execution is the difference between the intentions of the user and the allowable actions the system provides";
- The gulf of evaluation – "The gulf of evaluation reflects the amount of effort that the person must exert to interpret the physical state of the system and to determine how well the expectations and intentions have been met";

Some of Norman's stages correspond roughly to Foley and Van Dam's separation of concerns [Foley et al., 1990]. The user forms a conceptual intention, reformulates it into the semantics of several commands, constructs the required syntax, and eventually produces the lexical tokens.

II.1.3.Usability principles and rules

Many sources define general rules and principles for usability. Those principles and rules, also called usability heuristics, provide design guides for interactive system development and can be effectively used in discount usability engineering heuristic evaluation [Nielsen, 1993].

In a study of the impact of the explanatory power of usability heuristics, Nielsen performed a factor analysis of the scores of 249 usability problems with respect to a list of 101 usability heuristics [Nielsen, 1994]. From this statistical analysis the author derived a candidate set of 10 heuristics providing a broad coverage of the usability problems found in common interactive applications. This candidate set of heuristics includes the following principles:

H1. Visibility of system status – the system should always keep users informed about what is going on, through appropriate feedback within reasonable time;

H2. Match between system and real world – the system should speak the users' language with concepts familiar to the user in a logical and natural order, rather then system-oriented terms;

H3. User control and freedom – provide ways for the user to escape from unwanted states without going through extended dialogues. Support undo and redo;

H4. Consistency and standards – avoid different terms, situations and actions with the same meaning. Follow platform conventions, standards and guidelines;

H5. Error prevention – design the system carefully to prevent problems from occurring in the first place;

H6. Recognition rather than recall – make objects, actions and options visible. The user should not have to recall information from one part of the dialogue to another. Instructions for use of the system should be visible and retrievable whenever appropriate;

H7. Flexibility and efficiency of use – accommodate different experience levels providing accelerators for expert users that are invisible to novices. Allow users to tailor frequent actions;

H8. Aesthetic and minimalist design – avoid information in dialogues which is irrelevant and rarely needed, hence diminishing the visibility of relevant information;

H9. Helping users recognize, diagnose and recover from errors – error messages should be expressed in simple language, precisely indicating the problem and constructively suggesting a solution.

H10. Help and documentation – documentation and help, when required, should be focused on user tasks and easy, concise and easy to search.

In the following sections we present some of the most important usability principles found in the literature. The aim here is not to survey all of the published usability principles, but rather to complement Nielsen's candidate set of heuristics with design-oriented advice. For a complete review of usability heuristics for evaluation and design purposes refer to [Baecker, 1995]. We then relate those principles and rules with the candidate set of heuristics proposed by Nielsen.

II.1.3.1. Norman's Principles of Design for Understandability and Usability

Norman's seven stages of action can be used as design aids or heuristics, because they provide a basic checklist to ensure the gulfs of execution and evaluation are bridged [Norman and Draper, 1986]:

- Provide a good conceptual model (H2) – a good conceptual model allows users to predict the effects of their actions, what to expect and how to react when things go wrong. The conceptual model provided by the designers should result in a consistent and coherent system image;

- Make things visible (H1) – the users should have no problems perceiving the system state and the alternatives for action;
- The principle of mapping (H6) – the users should clearly determine the relationship between the actions and results, the controls and their effects and between the system state and what is visible;
- The principle of feedback (H1) – the users should receive continuous and informative feedback of the result of their actions;

II.1.3.2.Constantine & Lockwood Design Principles

Constantine and Lockwood provide a set of design principles that complement the general rules for usability described in section II.1.1 [Constantine and Lockwood, 1999].

- Structure (H2) - Organize the user interface purposefully, in meaningful and useful ways based on clear, consistent models that are apparent and recognizable to users;
- Simplicity (H7) - Make simple, common tasks simple to do, communicating clearly and simply in the user's own language and providing good shortcuts that are meaningfully related to longer procedures;
- Visibility (H1) - Keep all needed options and materials for a given task visible without distracting the user with extraneous or redundant information;
- Feedback (H1) - Keep users informed of actions or interpretations, changes of state or condition, and errors or exceptions that are relevant and of interest to the user through clear, concise, and unambiguous language familiar to users;
- Tolerance (H3 and H5) - Be flexible and tolerant, reducing the cost of mistakes and misuse by allowing undoing and redoing while also preventing errors wherever possible by tolerating varied inputs and sequences and by interpreting all reasonable actions reasonably;
- Reuse (H4 and H6) - Reuse internal and external components and behaviors, maintaining consistency with purpose rather than merely arbitrary consistency, thus reducing the need for users to rethink and remember;

II.1.3.3.Shneiderman's Eight Golden Rules of Interface Design

Shneiderman provides a set of eight golden rules for interface design derived heuristically from experience [Shneiderman, 1998]

- Strive for consistency (H4) – consistent sequences of actions, terminology, color, layout, capitalization, fonts, and so on, should be required in similar situations. Exceptions should be comprehensible and limited in number.

- Enable frequent users to use shortcuts (H7) – abbreviations, special keys, hidden commands and macro facilities should be used to reduce the number of actions required as the frequency of use increases.
- Offer informative feedback (H1) – provide system feedback for every user action. For frequent and minor actions provide modest responses and for infrequent and major actions provide substantial response.
- Design dialogs to yield closure (H1 and H3) – sequences of actions should be organized into groups with a beginning, middle and end. The informative feedback at the completion of a group of actions provides satisfaction of accomplishment.
- Offer error prevention and simple error handling (H5 and H9) – as much as possible design the system such that users cannot make serious errors. If the users make an error, the system should detect the error and offer simple, constructive, and specific instructions for recovery.
- Permit easy reversal of actions (H3) – as much as possible actions should be reversible, relieving anxiety and encouraging exploration.
- Support internal locus of control (H1 and H3) – provide users with the sense that they are in charge of the system and that the system responds to their actions.
- Reduce short-term memory load (H8) – conform to the seven-plus or minus-two chunks of information.

II.1.3.4.Usability Design Principles and the Candidate Set of Heuristics

Figure II.10 depicts the usability heuristics described in the previous section and their relationship with respect to Nielsen's candidate set of heuristics. The illustration suggests that there are five main clusters of heuristics corresponding to the following concerns:

- The importance of matching the user's conceptual model with the system image, thus leveraging on the previous experience the users acquired in the application domain. This clusters involves the heuristics related to providing good conceptual models, that yield well-structured user-interfaces based on clear mappings between the actions and representations of the user-interface;
- Visibility and feedback as major factors enabling users to perceive the system state and the alternatives for action through clear, concise and unambiguous language. This cluster involves the heuristics related to feedback, visibility and closure that enable the users to feel in control of the user-interface.
- Flexibility and efficiency enabling the system to accommodate different user experience levels. This cluster involves heuristics related to providing infrequent users with simple user-interfaces, while leveraging shortcuts for frequent users.
- Consistency as a way of avoiding different situations, actions and terms with the same meaning. This cluster involves heuristics related to following standards,

guidelines and platform conventions that should be reused with purpose to avoid arbitrary consistency.

- Error prevention, tolerance and reversibility of actions as a way to reduce the cost of mistakes, errors and misuse. This cluster involves heuristics related to error prevention, error handling, undoing, redoing and tolerance to varied input sequences.

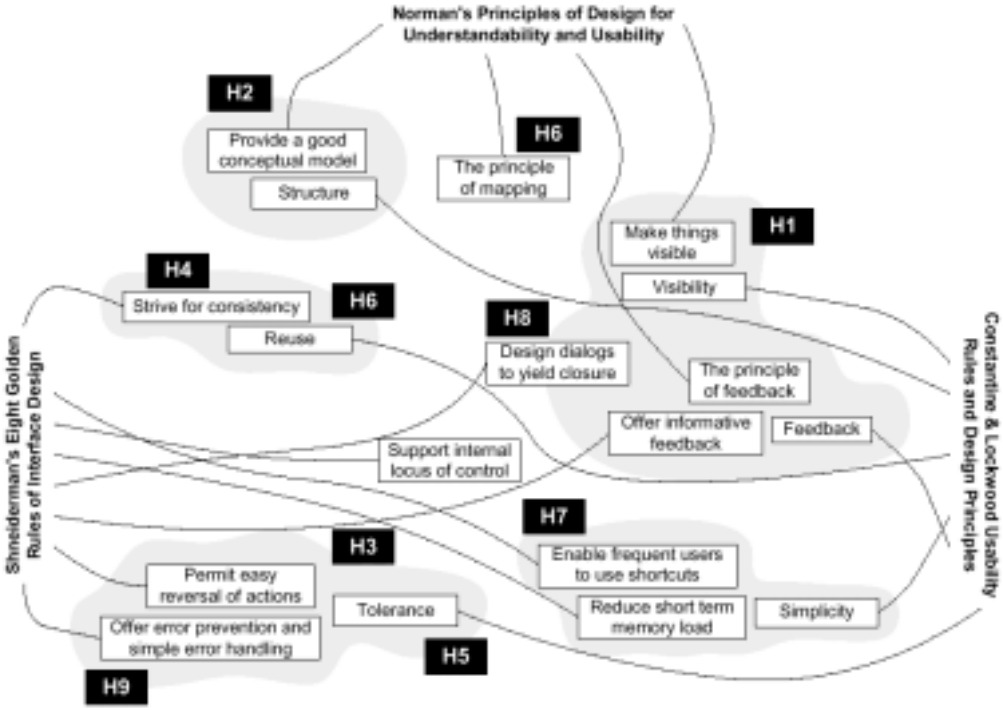


Figure II.10 – Relationship between usability heuristics

II.2. A BRIEF HISTORICAL PERSPECTIVE OF OBJECT-ORIENTED MODELING

Modeling in software development began with the use of flowcharts and related techniques to model procedures and algorithms for "traditional" programming languages, such as Cobol and Fortran. Modeling for analysis and design purposes emerged in the 70s with Structured Analysis and Structured Design [Yourdon and Constantine, 1979] and became widespread in the 80s, notably for large software development environments. Analysis and design models have since become fundamental in both teaching and practical experience in the fields of computer science and software engineering.

However there are many different ways to specify, document and implement software intensive systems. Traditional approaches based on programming languages or pseudo-programming languages, are popular for many situations but far from being the best and only way. Different modeling approaches go from rules and logic to neural nets, genetic algorithms, Petri nets, structured techniques, entity-relationship and many others. In the words of Martin and Odell: "there are many development approaches: there always will be and always should be (…) different tasks can have different characteristics that require different kinds of approaches" [Martin and Odell, 1998]. Object orientation is then a mechanism that organizes and interconnects many different development approaches, bringing coherence to the myriad that modern software based development faces. Object orientation is not an approach that models reality but the way people understand, process and manipulate reality [Martin and Odell, 1998].

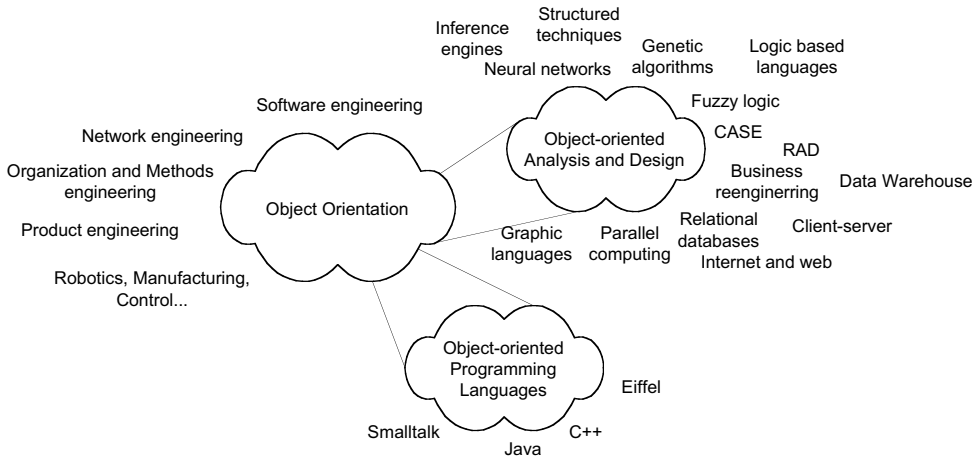


Figure II.11 – Object orientation, object-oriented analysis and design and object-oriented programming languages (adapted from Martin and Odell [Martin and Odell, 1998])

Object orientation is a technique for organizing knowledge, that is, to organize the way we think about the world. This organization is expressed in terms of types of things (object types), their attributes, operations, relationships, rules that drive their behavior and collaborations. Although object-orientation has been exclusively associated with programming languages, its scope is far broader than implementations of object-oriented principles and even of any kind of implementation technology. Another common misconception about object-orientation concerns its limitation to information systems. Although information is an important part of software intensive systems, there are also other concerns involved in developing moderns software system, such as business process engineering (including those that are not automated by software), organization and methods engineering, network engineering, hardware engineering and software engineering. Thus object orientation can be used effectively for engineering any kind of system. See Figure II.11 for some examples of applicability of object-orientation.

II.2.1.The nature and Importance of Modeling

A model is a representation in a certain medium of the important aspects of the thing being modeled with a certain purpose and from a certain point of view. Thus models simplify and enhance the ability to reason about the thing being modeled by omitting certain aspects. Moreover, models are represented in media, involving the semantics and notation, which is suitable for communicating the understanding of modelers in a way that is succinct and unambiguous.

Models of software intensive system encompass three major aspects: semantic information (semantics), visual presentation (notation) and context [Rumbaugh et al., 1999]. Semantics capture the meaning of the model and includes the syntactic structure, well-formedness rules and execution dynamics. In object-oriented modeling semantics are captured through a network of logical constructs (classes, associations, states, use-cases, etc.). The semantic elements are used to generate workable products, validity checking, metrics and so on. The visual presentation shows semantic information in a form that can be seen and manipulated by humans. Presentation elements form the notation and they are responsible for guiding the understanding of the model in a usable way. Presentation elements derive their own semantics from semantic model elements. However, they are not completely derived by semantics, the arrangement - and other aesthetic properties - can convey human meaningful semantic information that is too weak or ambiguous to be formalized in the semantic elements.

Models of software intensive systems are developed in a larger context that conveys their true meaning. This context includes the internal organization of the model that involves decomposition for manageability or cooperative work; annotations and other non-semantic information regarding the development process itself (for instance, versioning, permissions, etc.); a set of defaults, assumptions and customizations for

element creation and manipulation; and finally relationships to the environment in which they are used, including modeling and development tools and other development constraints (operating systems, project management, etc.).

Models in software development have many uses and advantages and serve different purposes. The following are some of the most important ones adapted from [Rumbaugh et al., 1999].

- To capture and precisely state the requirements and domain knowledge in order to foster understanding and agreement among the different stakeholders;
- To foster exploration of different design decisions, aiming at devising an accurate overall architecture before going into detailed design;
- To capture design decisions in a mutable form separate from the requirements, thus fostering separation of concerns between the external behavior (including the real world information involved) and the internals that implement them;
- To generate usable work products, even if not complete and executable, or to aid and guide the process of generating workable products;
- To organize, find, filter, retrieve, examine, and edit information about the systems using different views that project parts of the complete model for a certain purpose;
- To explore multiple solutions economically by permitting to deal with complexity that otherwise is difficult to master directly, thus enabling simulation, change impact, exploration of dependencies, etc.

Another important characteristic of models is that they can represent the system at different levels of abstraction. Abstraction plays a central role in modern software-based development. Abstraction allows modelers to defer thinking about details, thus fostering exploration and innovation. According to Rumbaugh and colleagues [Rumbaugh et al., 1999] the amount of detail represented in different models must be adapted to the following purposes.

- Models that guide the though process start at a high-level capturing requirements and enable exploration of different options and generation of ideas before converging on the right solution. Early models are ultimately replaced with more accurate models until the detail and precision of an implementation model is reached. Although the different versions of early models are not required to be preserved, whenever a model reflects a complete view at a given abstraction level, it should be preserved.
- Abstract specifications of the essential structure of the system reflect the key concepts and mechanisms of the eventual system and are intended to evolve to implementation models. Essential models focus on key semantic intent properties and traceability to final models must assure that those properties are correctly incorporated in the end-system.

- Models that fully specify a final system include not only the logical semantics of the system (including data structures, algorithms, etc.) but also organizational decisions about the system artifacts necessary for cooperative work by humans and processing tools.
- Exemplars of typical or possible systems give insight to humans and can validate system specifications and implementations. An example model includes instances rather than general descriptions.
- Models can represent complete or partial descriptions of systems. Partial descriptions of distinct, discrete units of a system are a common way to ensure reuse since they can be combined in different ways to form different systems.

A model can also be seen as a type of potential configurations of a system. The model extent (or set of instances) is thus the possible systems that are consistent with the originating model. The meaning (or intent) of the model is then the description of the generic structured capable of generating different instances. Models can vary in abstraction and detail, they can be prescriptive or descriptive and they are subject to different interpretations.

II.2.2.Object-Oriented Analysis and Design

Object-oriented analysis and design concerns the use of object orientation for the purpose of analyzing and designing software-based systems. In an attempt to formalize object-oriented analysis and design, Martin and Odell define analysis and design as follows [Martin and Odell, 1998]: "Analysis is a process that maps from a perception of the real world to a representation of that perception". Conversely the authors define, design as a "process that maps from an analysis representation to an expression of implementation". Both processes are different because the perception aims at understanding the problem, and design drives the eventual implementation of a solution.

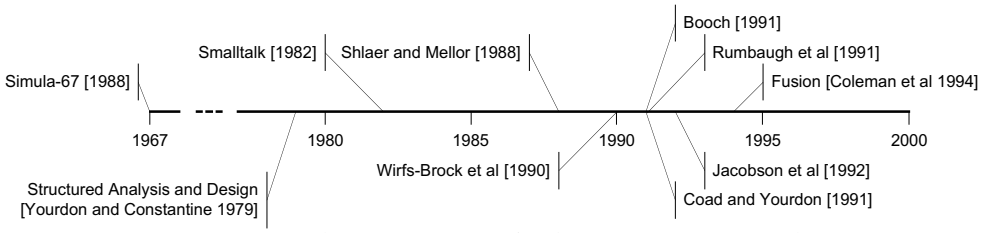


Figure II.12 – Historical Perspective of Object-Oriented Analysis and Design Methods

Object-oriented analysis and design methods emerged from the early efforts of using object orientation in programming languages. The first object-oriented language is generally acknowledged to be Simula-67. Although Simula didn't have a significant following, it largely influenced subsequent developments in the field – particularly enhancing the scope of object-orientation in software development. The object-oriented movement became active with the widespread availability of Smalltalk and

other object-oriented languages in the 80s. The impact of object-oriented languages as an implementation technology shifted the focus to development methods in the late 80s. The first methods were published in the late 80s and early 90s. Figure II.12 illustrates the major contributions for object-oriented analysis and design methods, from the initial proposal of Simula to the so-called second-generation methods that formed the basis of the Unified Modeling Language.

From the initial first-generation methods emerged a plethora of different extensions and new proposals that presented some new and interesting ideas, but also a multitude of similar concepts, notations, definitions and terminology. One of the justifications for this activity, in the early 90s, was the impact of object-orientation in software development as a means of coping with the increasingly complexity of modern software products. On the other hand, a good number of the first and second-generation methods included not only semantics and notation but also process issues (thus the name methods). Hence, most of the approaches were in fact object-oriented methods in the sense that they also provided specific guidance for carrying out a particular task (or set of tasks) in the context of software development. Furthermore, some of the approaches focused or where particularly strong on different application domains (for instance real-time systems) or specific phases in the development lifecycle (for example analysis or design).

II.2.3.The Unification Effort

In the middle 90s the idea of unifying different approaches emerged and was first suggested by Booch in a meeting at OOPSLA'93 [Martin and Odell, 1998]. Although the initial efforts toward standardization failed, the successive hiring of Booch, Rumbaugh and Jacobson for Rational Corporation enabled the development of what was first known as the unified method (presented in 1995 at OOPSLA). By the same time the Object Management Group (OMG) revived a group known as the Object Analysis and Design Task Force (OA&DTF) which issue the first Request for Proposal (RTF) in June 1996 for what is currently known as the Unified Modeling Language (UML). The initial RTF was the first step towards a standard, non-proprietary and general-purpose object-oriented modeling language. In the requirements of the RFP were included: a meta-model representing the semantics of models; an interface description language (IDL) for tool interoperability; a set of notations for representing models; and finally a mechanism to accommodate evolution of models and incorporation of additional semantics.

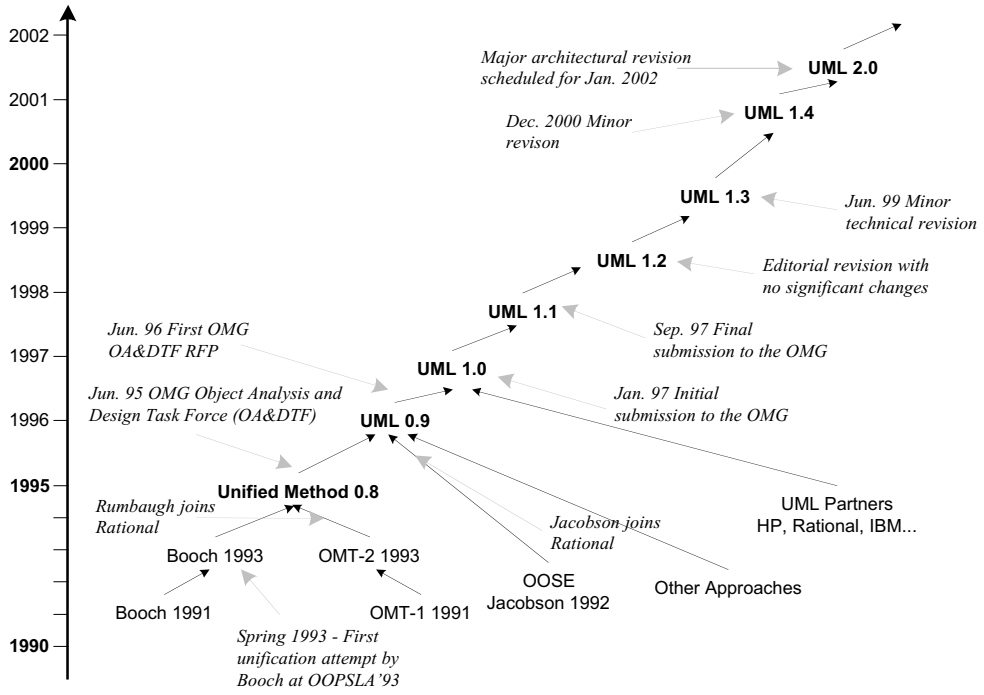


Figure II.13 – UML Genealogy and Roadmap (sources: [Kobryn, 1999] and [Martin and Odell, 1998])

Figure II.13 illustrates the evolution of the Unified Modeling Language from the early first and second-generation methods to the proprietary unification efforts and the subsequent standardization under the OMG's supervision.

II.2.4.Martin and Odell Foundation of Object-Oriented Modeling

Martin and Odell provide a foundation of object-oriented modeling consisting of the ideas that are fundamental to capture the way people think and understand reality. This approach is based on the claim that the better way to specify system requirements' is to document the way people think about the system.

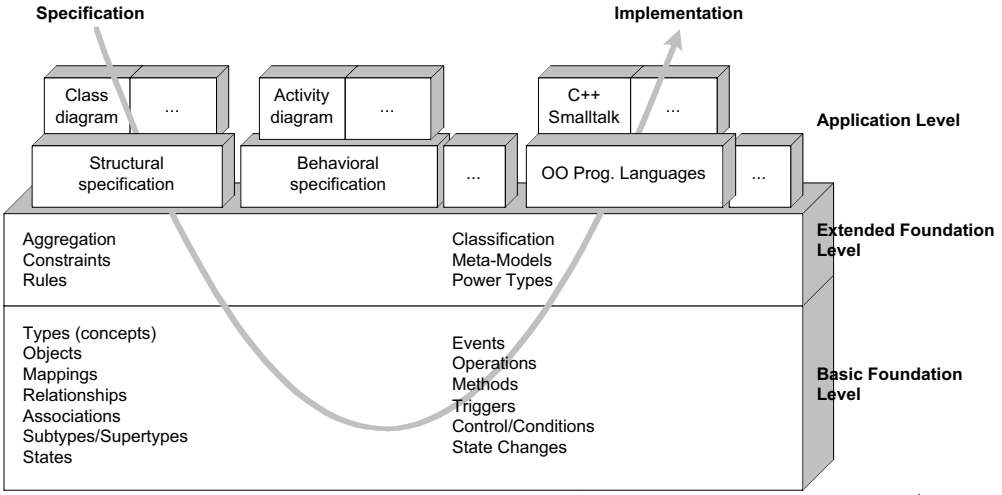


Figure II.14 – Martin and Odell Foundation of Object-Oriented Modeling (adapted from Martin and Odell [Martin and Odell, 1998]

Figure II.14 illustrates the foundation for object-oriented modeling. The framework is divided into three levels, and aims at providing a formal foundation that provides a way to precisely and unambiguously build systems that perform consistently and correctly.

The basic foundation level consists of the ideas that are fundamental to thinking about the structure and behavior [Martin and Odell, 1998]. Structure refers to a static and spatial metaphoric vision of how objects are laid out in space. Structured can be used to specify various object configurations. In contrast, behavior refers to how the world changes over time. Behavior can be used to specify the processes that query or modify objects. The blocks in the basic foundation are independent of notation, thus they can be represented in graphical or textual form. Furthermore, they can be used to specify structure or behavior from a conceptual or implementation standpoint, independent of the techniques used to express the implementations. For instance, programming languages, database definition languages or data manipulation languages.

The extended level is constructed from the twelve basic blocks of the foundation level and extends them to include more advanced structures that provide a way to describe and specify the system with increased ease and power [Martin and Odell, 1998].

The application level is where we make use of the foundation through representations of the structures in the previous levels. The representations can appear at the specification or implementation levels and include different notations, programming languages and other ways to communicate the basic and extended constituents of the foundation [Martin and Odell, 1998].

II.3. THE UNIFIED MODELING LANGUAGE

The Unified Modeling Language is a language for visualizing, specifying, constructing and documenting software intensive systems. In a short period of time the UML emerged as the dominant modeling language in the industry. It has found widespread use in many different application domains, from avionics and real-time systems to e-commerce and computer games. It applies to the different modern requirements to build and document software-based systems. It is used for business process modeling, to describe software architectures and patterns, and maps to many different middleware architectures, frameworks and programming languages. The UML is also recognized to have influenced the CASE tool industry, improving the quality and availability of high-end case tools, in particular for modeling purposes.

The word unified has several relevant meanings for the UML [Rumbaugh et al., 1999]. The UML combined the commonly accepted concepts from many first and second-generation object-oriented analysis and design methods (including those described in section II.2.2). The UML is also seamless across the development lifecycle, from requirements to deployment, in a way that is consistent with the object modeling foundation depict in Figure II.14. Besides it interconnects internal concepts in a way that permits representing the underlying relationships capturing them in a broad way applicable to many known and unknown situations. In addition, the UML successfully supports different application domains. Although special-purpose languages are still useful in many circumstances, the UML incorporates the interconnection characteristic illustrated in Figure II.11. Furthermore, the standard language has proved capable of supporting different implementation languages and platforms, including programming languages, databases, fourth-generation languages (4GLs), firmware, middleware, and so on. This characteristic is again consistent with the application level of the foundation framework presented in section II.2.4. Finally the UML is detached from any methodological background, that is, it doesn't include a detailed description of a specific development process.

II.3.1.Goals and Definition of the UML

There were a number of goals behind the development of the UML. Some of them are present in the original (and subsequent) RTFs issued by the OMG as mentioned in section II.2.3. Furthermore because the UML is an OMG standard it is nonproprietary and, hence, can be used and implemented freely by developers and vendors.

The primary goals of the UML, according to the OMG [OMG, 1999], are as follows:

- To provide users with a ready-to-use and expressive visual modeling language to exchange and develop meaningful models based on a common and familiar set of concepts;
- To provide extensibility and specialization mechanisms to extend the core concepts enabling tailoring for newly discovered needs or specific application domains;
- To support specifications that are independent of particular implementations technologies and development methods or processes;
- To provide a formal basis for understanding the language that is both precise and approachable. This is achieved using meta-modeling facilities and expressing the operational meaning in both natural language and a formal Object Constraint Language (OCL);
- To encourage the growth of the object tool market by providing a common set of concepts that can be interchanged between tools and users without loss of information;
- To support higher level development concepts - such as frameworks, components and patterns – supporting the full benefit of object modeling and fostering reuse;
- To integrate best practices in the domain of object modeling, like different views based on levels of abstraction, domains, architectures, life cycle states and so on.

The UML is defined by a set of documents controlled by the OMG and containing the following sections [Cook, 2000]:

- UML Summary – overview of the language, history, acknowledgements, etc.
- UML Semantics – definition of the UML abstract syntax in the form of a set of UML packages. The language is defined using a four-level meta-modeling facility (see section II.3.2). An English description is provided for each class in the meta-model and well-formdness rules are provided for each model in OCL.
- UML Notation Guide – describes the pictorial elements that make up the UML diagrams and how they fit together to form the UML notation. Mappings between the notation and the meta-model are provided.
- UML Standard Profiles – a brief introduction to two standard profiles (variants of the UML): the UML profile for software development processes, and the UML profile for business modeling.
- UML Corba Facility Interface Definition – defines an interface to a repository for creating, storing and retrieving UML models in the CORBA Interface Definition Language (IDL).
- UML XMI DTD Specification – the XML Metadata Interchange (XMI) defines an alternate physical specification of the UML in eXtensible Markup Language (XML) specifically for interchanging UML models between tools and/or repositories.
- Object Constraint Language Specification – introduces and defines the OCL that is used to formulate well-formdness rules in the UML specification.

The current definition of the UML is recognized to have several problems, in particular, regarding the precise definition of the semantics. According to Cook [Cook, 2000] the semantics of the UML is informally specified and the existing specification is scattered in a number of places. The UML semantics contains English prose and additional information in OCL about the meaning of the different constructs. Since English prose is subject to different interpretations, it is not possible to enforce or check a particular mapping of the UML for a specific purpose. Furthermore, the UML definition is uneven at the level of abstraction at which it defines semantics. The meta-model also contains significant redundancy, introducing concepts that are identical from the well-formdness perspective and that only differ in the English description and notation. Finally, the UML defines a set of diagram types but doesn't give a functional mapping between the meta-model and the notational elements, hence, compromising diagram interoperability at the syntactical level.

II.3.2.The UML Metamodel

UML is defined using meta-modeling facilities of a four-level architecture (depict in Figure II.15) where the relationships between layers are of the type *instance-of*. The four-layers of the OMG architecture are as follows.

- M3 – Meta-meta model or Meta object facility (MOF) – defines the basic concepts from which specific meta-models are created at the meta (M2) level. The MOF also contains a set of rules that specify the interoperability semantics and interchange format for any of its meta-models. M3 is defined using itself; hence, obviating the need for subsequent levels (M4, M5, etc.).
- M2 – Meta-model – defines the different specific meta-models of the OMG standards, which share the common meta-object facility. Examples of such meta-models are the UML, the Common Warehouse Model (CWM), Enterprise Java Beans (EJB), and so on.
- M1 – Model of the domain specific information – instances of domain specific information reside at M1 and usually correspond to class instances (or objects) or runtime data.
- M0 – Domain specific information – is where the users' models reside by instantiation of the different technology standards defined at M2. For instance, the model of a specific accounting software system in the UML and a potential implementation model of the application in EJB, both reside in M0.

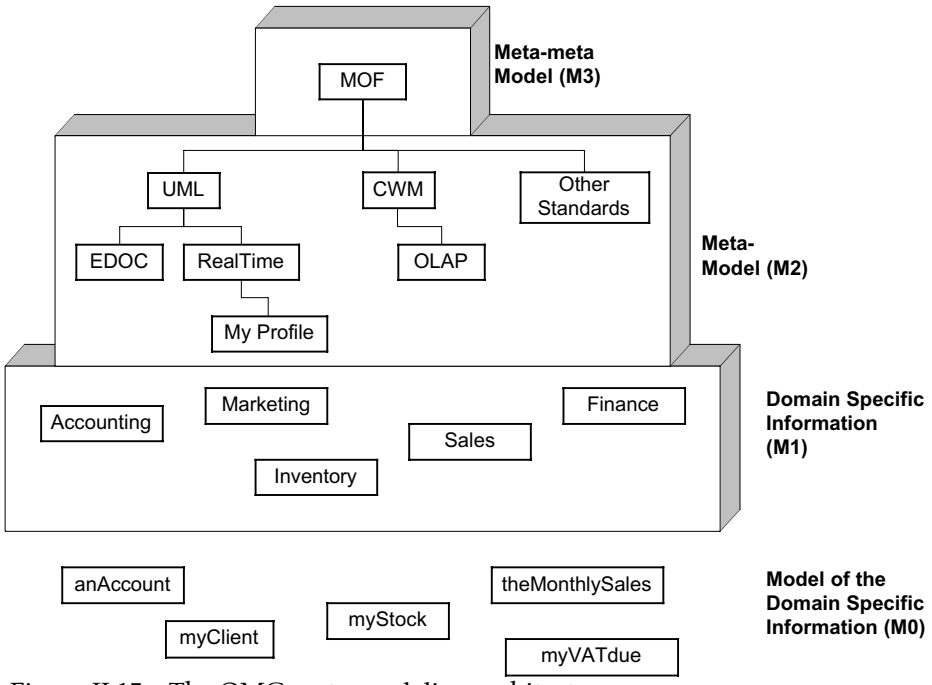


Figure II.15 – The OMG meta-modeling architecture

The MOF, and the corresponding OMG 4 layer architecture, bridge the gap between dissimilar meta-models by providing a common basis for interoperability and interchange. Furthermore, if different standards are MOF compliant they can reside in the same repository.

II.3.3.Extension Mechanisms and Future Development for UML Variants

Version 1.4 of the UML only provides two techniques to define UML variations. Profiles are defined in the UML standard as a predefined set of stereotypes, tagged values, constraints and notation icons that collectively specialize and tailor the UML for a specific domain or process [OMG, 1999].

A UML stereotype is a way of classifying UML elements as if they were instances of new meta-model concepts. This technique effectively provides a way of extending the UML meta-model at the M2 level (as illustrated by *myProfile* in Figure II.15). A stereotype may specify additional constraints and tagged-values that apply to its instances (residing at the M1 level) [Cook, 2000]. A stereotype is commonly used to indicate a difference in meaning or usage between two elements with identical structure. A UML tagged value is a pair (tag and value) that allows arbitrary information to be attached to any UML model element. A UML constraint enables new semantics to be specified linguistically (whether in natural language or OCL) to different model elements [Cook, 2000].

Collectively the three concepts described above offer a means of extending the UML with new kinds of modeling elements. The profiling extension mechanism is usually considered a lightweight extension mechanism because tools easily support it without

requiring "meta-tool" facilities, and also because it doesn't impact the interchange formats [Cook, 2000]. This so-called lightweight extension mechanism is recognized to be insufficient as a general way of extending the UML, because every new concept introduced must be represented either directly by an existing UML M2-class or by stereotyping an existing UML M2-class. Notably, the extension can only occur if a UML M2-class provides the basis for the new concept and if such extension is consistent with existing UML semantics [Cook, 2000].

A more heavyweight extension mechanism to the UML is provided, by using the OMG's meta-object facility (MOF) directly to create additional concepts at the M2-level. This mechanism is actually the approach taken by the OMG to define other standards such as the CWM and the EJB depict in Figure II.15. This approach is significantly more flexible than the existing profiling mechanism since it enables new M2-classes to be provided as subclasses of the UML M2-classes or as new M2-classes with associations to the UML definition. However there are a number of drawbacks, as pointed out by Cook [Cook, 2000]. Remarkably, the UML meta-model is not yet MOF compliant and there are no rules regarding the notational mappings and the semantics when the meta-model is extended.

To overcome the mentioned problems, with existing UML extension mechanisms, several authors have new approaches for extension facilities. Cook et al [Cook et al., 1999] proposed the concept of UML prefaces as a more general extension mechanism. A preface is a document that defines a UML extension by specializing, tailoring and extending collectively the meta-model, the abstract syntax, the semantics, the concrete syntax and the allowed transformations and generations. According to the authors, the adoption of a preface extension mechanism would be a substantial wide-ranging instrument covering a wide variety of subject matters. For a complete list of the coverage of the UML preface proposal refer to [Cook et al., 1999].

Atkinson and Kuhne proposed another approach that overcomes the obligation of defining modeling elements solely at the meta-model level (M2) through meta-instantiation. This approach, known as strict profiles, is consistent with the rules of strict meta-modeling and enables the placement of model elements at the *instance-of* hierarchical level they most naturally fit. Strict profiles enable the usage of regular M1-level inheritance, as well as meta-instantiation, and enables users to build upon the existing set of predefined constructs. Hence, strict profiles provide a way to distribute elements in the meta-modeling architecture avoiding some of the problems with semantic distortions and inflexibility with the current "meta=predefined" principle that underlines the UML standard [Atkinson and Kuhne, 2000].

II.4. OBJECT-ORIENTED PROCESS AND METHODS: THE UNIFIED PROCESS

As we saw in the previous sections the UML is detached from any methodological background, that is, it doesn't provide any specific guidance to a software development process or lifecycle. The methodological directions, methods and techniques involved in the originator methods of the UML - in particular those proposed by Booch, Rumbaugh and Jacobson - are now included in the Unified Process [Jacobson et al., 1999]. Other methodological contributions, apart from notable exceptions, pertain as individual techniques or methods that have been adapted to the UML notation.

A software process is defined by Fuggetta [Fuggetta, 2000] as the coherent set of policies, organizational structures, technologies, procedures and artifacts required to conceive, develop, deploy and maintain a software product. This broader vision of software development as a process is different from that of a software lifecycle, which defines the different stages in the lifetime of a software product.

A software process involves a number of different contributions and concepts, as follows [Fuggetta, 2000]:

- Software development technology – the technological support used in the process, for instance, tools, infrastructures, etc.
- Software development methods and techniques – the guidelines on how to use technology to accomplish software development activities;
- Organizational behavior – the coordination and management of people within the organizational structure;
- Marketing and economy – the context in which the software must be sold and used, for instance, the customers, the market settings, etc.

Viewing software development as a process, and clearly distinguishing the lifecycle stages, has helped identify the different problems that need to be addressed in order to establish effective software development practices. Those practices include the activities that need to be accomplished to achieve the process goals; the roles that people play in the process; the structure and nature of the artifacts that need to be created and maintained; and finally the tools to be used.

The Unified Process is an UML-based process framework that originated from the initial Ericsson approach by Jacobson and colleagues in the 70s. The Ericsson approach introduced the ideas of software interconnected blocks (subsystems and

components in the UML) and traffic cases (use-cases in the UML). Those fundamental ideas were improved in the 80s with the Objectory commercial software process, also known for the name of the seminal book from Jacobson and colleagues Object-Oriented Software Engineering (OOSE) [Jacobson, 1992]. Objectory established the concept of use-cases driving development and supporting traceability throughout the lifecycle. The workflows defined in the Objectory process supported different models (requirements, analysis, design, implementation and test), corresponding to different stages in the lifecycle.

The Rational Objectory Process emerged with the acquisition of Objectory by Rational. The initial concepts of the Objectory process merged with the architecture-centric and iterative incremental nature of the Rational Approach. The architecture-centric style of development was based on a representation involving 4+1 views: the logical view, the process view, the physical view and the development view, plus an additional view that illustrates the first four views with use-cases and scenarios [Kruchten, 1995]. The iterative and incremental nature was based on a four-phase model (inception, elaboration, construction and transition) devised to better structure and control progress. The Unified Process, and its companion commercial version (the Rational Unified Process), gained their final designation with the advent of the UML in the late 90s. The UML is used to represent all the models in both versions of the process; it is also used as a process modeling language (PML) to describe the software development workflows involved.

II.4.1.Use-case Driven, Architecture-centric, Iterative and Incremental

The Unified Process is considered use-case driven to denote the emphasis on knowing and understanding what real users want and need to accomplish with the envisioned system.

The term user in the UP refers to, not only real users, but also other systems. The actual description is "the term user represents someone or something (…) that interacts with the system being developed" [Jacobson et al., 1999]. The users, as described in the UP, interact with the system through a sequence of actions that provide a result of value. A sequence of meaningful actions is considered a use case, which the authors of the UP describe as "a piece of functionality in the system that gives a user a result of value" [Jacobson et al., 1999]. A use-case model is then the collection of all the use-cases that describe the complete functionality of the system.

Hence, the use-case driven nature of the UP, is grounded on two basic assumptions. On the one hand, the use-case strategy forces developers to think in terms of real value to the users and not just in terms of functionalities. On the other hand, use-cases drive the development process because they are used to understand the requirements, they drive the development and review of the analysis and design level models, and

they also facilitate testing of the implementation components for conformance with the requirements.

While use-cases drive development, their selection process should take place in the contexts of the overall system architecture. This characteristic is known in the UP as architecture-centric. The software architecture, in UP terms, refers to the most significant static and dynamic aspects of the system. The architecture grows out of use-cases but involves many other factors, such as platform restrictions, the existing reusable components, deployment considerations, legacy systems, and so on. Use-cases and the architecture correspond to the function and form of the software system and must evolve in parallel. The use-cases define the system functionalities and the architecture focuses on understandability, resilience to future changes and reuse.

The final main characteristic of the Unified Process concerns the iterative and incremental development strategy. In the UP the overall development process is divided into several mini-projects. Each mini-project comprises a selection of use-cases for implementation and deals with the different risks involved. Hence, mini-projects correspond to controlled and planned iterations that result in increments to the end product [Jacobson et al., 1999]. The goal of the iterative and incremental development strategy in the UP is to reduce the risks involved in single increments, thus, increasing the likelihood of achieving the planned budget and time to market.

II.4.2.The Unified Process Lifecycle

The lifecycle of the Unified Process consists of four phases per cycle, with each cycle ending with a product release. The four phases defined in the UP are as follows [Jacobson et al., 1999]:

- Inception – this phase defines the scope of the project and develops the vision of the end product and it's business case. During this phase a simplified use-case model containing the most critical use-cases is built and a tentative architecture outlined. The most important risks are also identified and prioritized leading to rough estimate of the overall project.
- Elaboration – during this phase most of the product's use-cases are specified in detail and the system architecture is designed. During elaboration architecture is expressed in terms of views of all the models of the system [Kruchten, 1995] and the most critical use-cases are realized leading to the architecture baseline. At the end of this phase the project manager should be able to plan the activities and estimate the resources required to complete the project.
- Construction – during this phase the product is built from the architecture baseline developed in the previous phase. Construction encompasses the bulk of development to implement all the use-cases agreed for the next candidate release. During construction minor architectural adjustments can happen and the developed product can still contain minor deficiencies.

- Transition – this phase covers the period from the initial tests with end-users (commonly known as beta release) to the actual deployment for the larger user community. The main activity during this phase is concentrated on reporting and correcting deficiencies in the product. However other activities such as manufacturing, training, help-line assistance and post-delivery defect correction happen during transition.

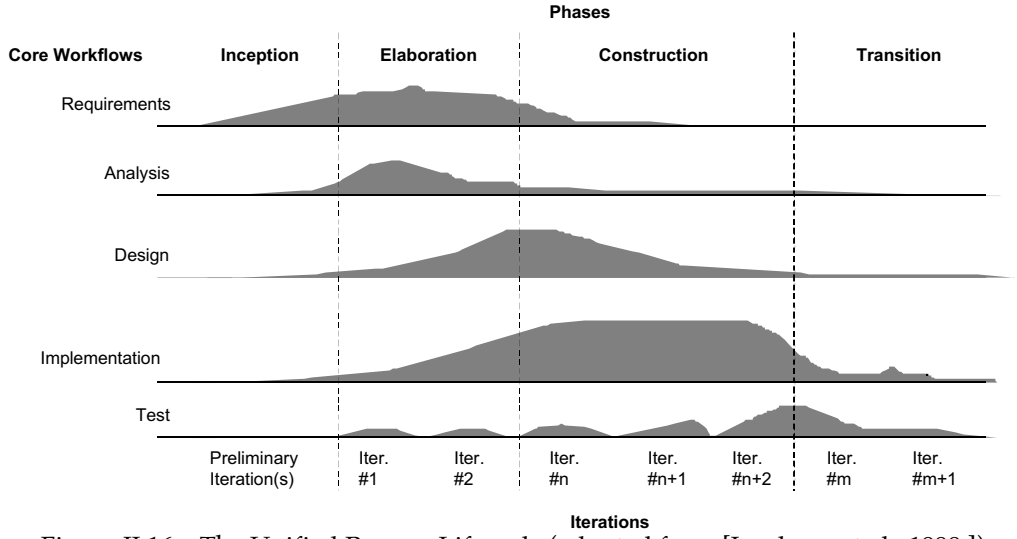


Figure II.16 – The Unified Process Lifecycle (adapted from [Jacobson et al., 1999])

Figure II.16 illustrates the Unified Process software lifecycle. The figure shows the development lifecycle in terms of the software development phases described above and also the core workflows and effort involved as time goes by. The iterative nature of the process is evident from the different iterations depicted in the time axis.

II.4.3.Architectural Representations in the Unified Process

One of the more important goals in software development is to establish the architectural foundation of the envisioned system. According to [Shaw and Garlan, 1996] a software architecture involves: "the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns".

Software architecture is represented by software architecture descriptions. Shaw and Garland [Shaw and Garlan, 1996] proposed an approach for architectural representation based on a generic ontology of seven entities: components, connectors, configurations, ports, roles, representations and bindings. The basic elements in the proposed ontology (depicted in Figure II.17) are: components that represent the loci of computations; connectors that represent interconnection between components; and configurations that define topologies of components. The remaining elements appear by the need to express interfaces in components and connectors. A component interface is defined by a set of ports, which define the point of interaction with its environment. Conversely a connector interface is defined by a set of roles, which

identify the participants of the interaction. Finally representations enable the description of the contents of a component or connector, required to represent hierarchical architectural descriptions.

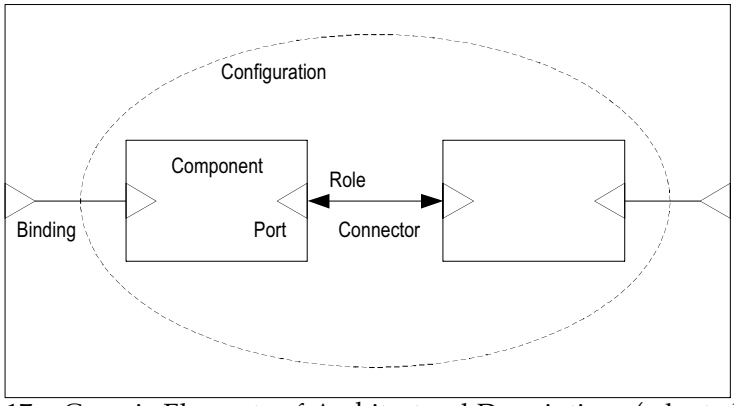


Figure II.17 – Generic Elements of Architectural Descriptions (adapted from Shaw and Garlan [Shaw and Garlan, 1996])

The concepts proposed in the previous ontology are easily mapped into UML descriptions because of its inherent syntax based on nodes and arcs. Therefore, many architectural representations can be described using UML diagrams, views and models. An UML diagram is defined in the standard as a graphical representation of a collection of model elements often rendered as a connected graph of arcs (relationships) and vertices (other model elements) [OMG, 1999]. If a UML diagram (or a set of diagrams) conveys a coherent abstraction of the system for a certain purpose, it is considered a model (see section II.2.1). The UML also supports the notion of view, defined as a projection of a model from a given perspective or vantage point, often-omitting entities that are not relevant to the given perspective [OMG, 1999].

II.4.3.1.The 4+1 View Model of Architecture

The Unified Process and its commercial version (the Rational Unified Process) suggest a five-view approach of the overall system architecture (depict in Figure II.18). This model, originally proposed in [Kruchten, 1995], defines five views as follows [Kruchten, 1998]:

- The logical view of the architecture addresses the functional requirements of the system. It is an abstraction of the design model and identifies the major design packages, subsystems and classes.
- The implementation view describes the organization of the static software modules (source code, data files, components, executables, etc.) in the development environment, both in terms of the packaging and layering and in terms of configuration management.
- The process view addresses the concurrent aspect of the system at runtime (tasks, threads and processes) and their interactions. It deals with issues of concurrency,

parallelism, system startup and shutdown, fault tolerance, object distribution, throughput, response time and so on.

- The deployment view shows how the various runtime components are mapped to the underlying platforms or computing nodes. It addresses the overall interconnection with system engineering and concerns like deployment, installation and performance.
- Finally the use-case view, in conformance with the use-case driven nature of the UP described in section II.4.1, contains the description of the main functionalities the system should provide to the end-users. The use-case view binds all the other views in the architecture and is used to validate them.

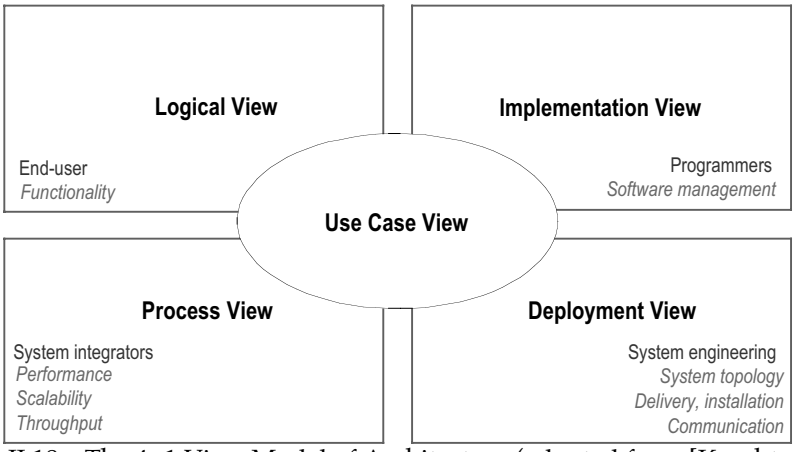


Figure II.18 – The 4+1 View Model of Architecture (adapted from [Kruchten, 1995])

The 4+1 view model of architecture is not completely embraced in the Unified Process as it is in the Rational Unified Process. Instead, the authors of the UP claim that the set of views in the 4+1 model align well with the major architectural descriptions found in the UP (the use-case, analysis, design and implementation models). Since a model is a complete description of the system from a given perspective and for a given purpose, the corresponding views, that describe the UP overall architecture, include only the architectural significant elements in the models. Hence, the major architectural description in the UP is the collection of views that contain the set of analysis, design, deployment, implementation and test elements which implement the architectural significant use-cases described in the use-case view.

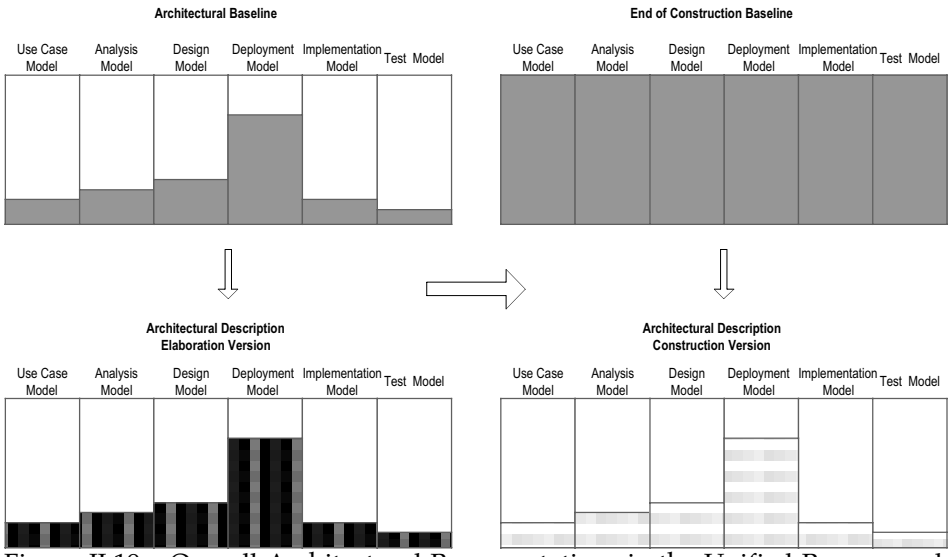


Figure II.19 – Overall Architectural Representations in the Unified Process and the Rational Unified Process

Figure II.19 illustrates one possible evolution of the different models of the UP that result from the elaboration phase. The architecture description is an extract (view) of the different models in the architecture baseline. The figure illustrates the completion process of the architecture baseline during construction (from upper left to right). The architecture description (from lower left to right) however doesn't change significantly since most of the architecture was defined during the elaboration phase. Only a few minor changes should occur (illustrated by the fill color) [Jacobson et al., 1999].

II.4.3.2.Architectural Patterns in the Unified Process

Besides the previous architectural representations, that define the overall system architecture, patterns also play a central role in the model-driven perspective of the Unified Process.

Patterns emerged from the ideas of the architect Christopher Alexander and are used to systematize important principles and pragmatics in the construction field. Those ideas have inspired the object-oriented community to collect, define and test a variety of solutions for commonly occurring design problems. Software patterns follow the same principle defined by Alexander, "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, without ever doing it the same way twice" [Alexander et al., 1977]. A software pattern is hence, a general solution for commonly occurring design problems.

The Unified Process defines patterns as collaboration templates that can be specialized for different purposes. One notable example of an architectural pattern is the boundary-control-entity framework used in the analysis model to structure the

requirements defined in the use-case model. This architectural framework, initially proposed in the Object-Oriented Software Engineering (OOSE) approach [Jacobson, 1992], divides analysis classes - into information, behavior and interface - to promote a structure more adaptable to changes. The underlying goal of this partition is that, assuming that all systems change, stability will occur in the sense that changes affect (preferably) only one object in the system (they are local) [Jacobson, 1992]. Therefore, the analysis framework concentrates changes to the interface and related functionality in interface (boundary) objects; changes to (passive) information and related functionality in entity objects; and changes to complex functionality (e.g., involving multiple objects) in control objects.

This approach is conceptually similar, although at a different granularity level, to the PAC model (see section II.6). In fact, the PAC distinction between presentation, abstraction (application) and control relate, conceptually, to the interface, entity and control objects.

The information space for the OOSE analysis framework defines three dimensions (depicted in Figure II.20):

- The information dimension specifies the information held in the system in both short and long term – the state of the system;
- The behavior dimension specifies the behavior the system will adopt – when and how the system's state changes;
- The interface dimension specifies the details for presenting the system to the outside world.

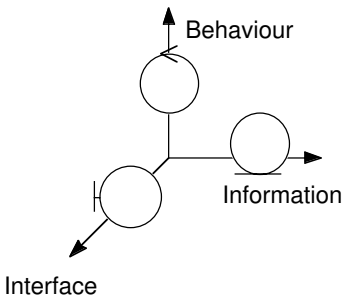


Figure II.20 - The information space of the OOSE analysis framework

With the advent of the UML, the Unified Process adopted the OOSE analysis framework. The influence of this approach can also be found in the UML standard, in the form of an extension profile for software development processes [OMG, 1999]. In this evolutionary process subtle conceptual and notational changes occurred. The original interface dimension and the interface object are known in the Unified Process as, respectively, presentation dimensions and boundary class stereotype.

II.5. USABILITY AND SOFTWARE DEVELOPMENT

Methodological work about usability engineering and the software development process started in the 1970s. One of the first references to usability engineering methodology was provided by Gould and Lewis [Gould and Lewis, 1985] and described an approach involving three global strategies: early focus on users and tasks, empirical measurement, and iterative design. Since this initial approach others have provided general frameworks for usability engineering [Nielsen, 1993; Shneiderman, 1998]. According to Button and Dourish [Button and Dourish, 1996] those approaches can be distinguished into three categories, regarding their integration with the software development process [Mayhew, 1999]:

- Introducing usability experts to design teams with the aim of providing input to the development process;
- Introducing usability engineering methods and techniques that are able to produce specific work artifacts to the development process;
- Redesigning the whole development process around usability engineering expertise, methods and techniques;

From the early focus on introducing usability experts and specific techniques and methods to the development process, a shift to redesign approaches occurred. This shift is consistent with the acceptance that usability engineering requires an active involvement of both usability experts and corresponding methods and techniques throughout the development lifecycle – from early inception to transition.

II.5.1.The Usability Engineering Lifecycle

The usability engineering lifecycle (UEL) Mayhew [Mayhew, 1999] is one of the most recent and complete approaches consistent with the principle of redesigning the software development process to integrate usability engineering. The UEL consists of several types of tasks, as follows [Mayhew, 1999]:

- Structured usability requirements analysis;
- Explicit usability goal setting task, driven directly from requirements analysis data;
- Tasks supporting a structured top-down approach to user-interface design driven directly from usability goals and other requirements data;
- Objective usability evaluation tasks for iterating design towards usability goals;

The usability engineering lifecycle is divided into three phases each one involving a set of tasks [Mayhew, 1999]. The lifecycle is illustrated in Figure II.21, where dependencies between tasks are depicted.

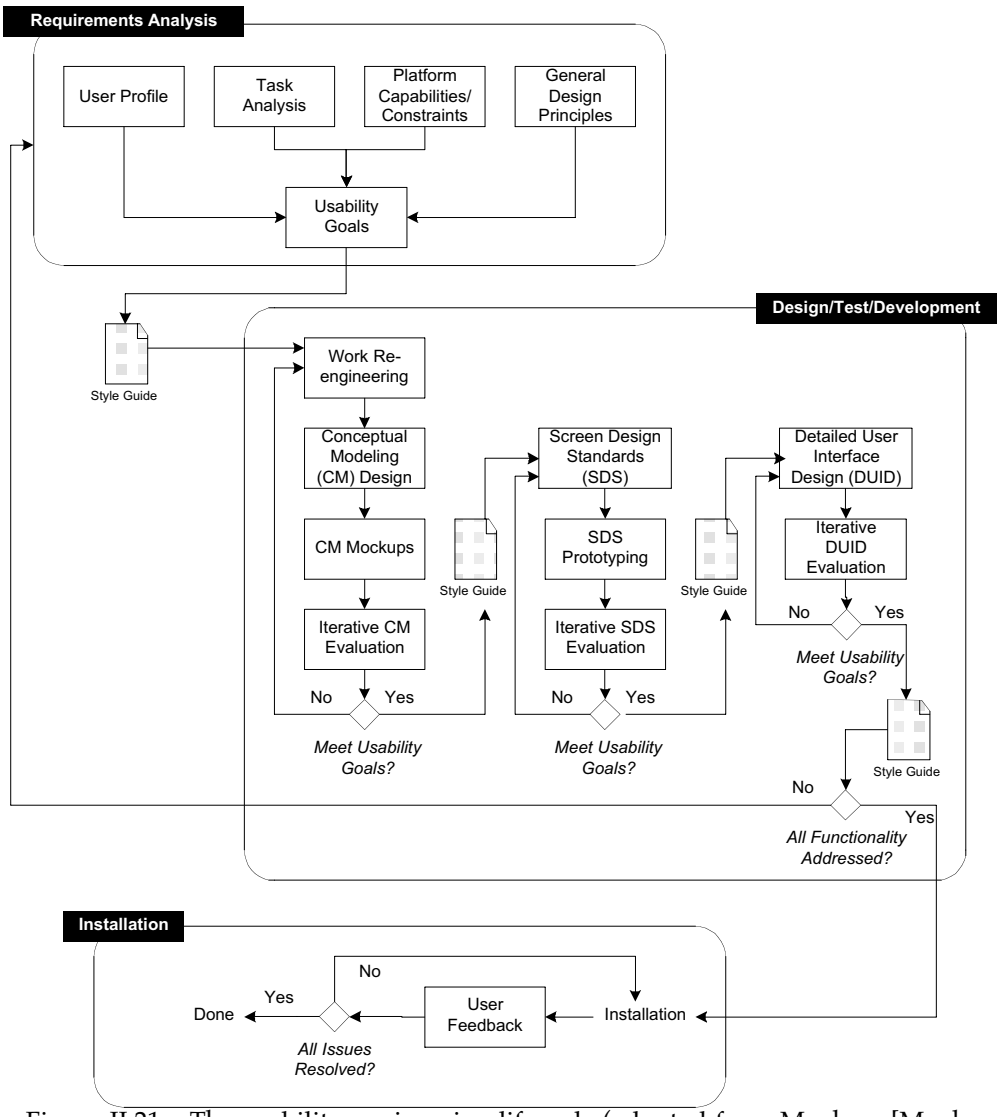


Figure II.21 – The usability engineering lifecycle (adapted from Mayhew [Mayhew, 1999]

Phase one is called requirements analysis and involves the following tasks:

- User profile – involves obtaining a description of the specific user characteristics relevant for user interface design from the intended user population;
- Contextual task analysis – involves studying the users' current tasks, workflow patterns and conceptual frameworks with the aim of producing a description of current tasks and workflows that underlie the user goals;
- Usability goal setting – this task involves defining qualitative goals reflecting usability requirements extracted from the previous tasks; and also quantitative goals defining minimal acceptable user performance and satisfaction criteria based on a subset of high-priority qualitative goals;

- Platform capabilities and constraints – involves determining and documenting the user-interface capabilities and constraints that define the scope of possibilities for user-interface design;
- General design principles – involves gathering and reviewing relevant user interface design principles and guidelines.

Phase two is called design/testing/development and divided into three levels each one dealing with different design issues. Level 1 is concerned with high-level design issues and contains four tasks:

- Work reengineering – involves redesigning users tasks at the level of the organization and workflow to streamline work and exploit the capabilities of automation;
- Conceptual Model Design – involves generating high level design alternatives such as navigational pathways, major display and rules for consistently presenting work products, processes and actions;
- Conceptual Model Mock-ups – involves generating paper-and-pencil prototype mock-ups of high-level design ideas about organization and conceptual model design;
- Iterative Conceptual Model Evaluation – involves the evaluation of the prototype mock-ups, through iterative evaluation techniques such us formal usability testing, and of real-users attempting to perform representative tasks with minimal training and intervention.

The second level of the design/testing/development phase involves four additional tasks as follows:

- Screen design standards – involves the development of a set of product-specific standards and conventions, for all aspects of detailed screen design, and based on mandated industry and/or corporate standards;
- Screen design standards prototyping – involves applying the screen design standards, and the conceptual model design to the detailed design of selected subsets of product functionality, implemented as a prototype;
- Iterative screen design standards evaluation – involves evaluating the screen design standards prototype, with an evaluation technique such as formal usability evaluation, and then iteratively redesigning and re-evaluating until all the major usability problems are solved;
- Style guide development – involves capturing in a document the set of standards and conventions validated and stabilized in the previous tasks.

Finally level 3 of the design/testing/development phase involves three tasks, as follows:

- Detailed user interface design – involves designing the complete product user interface based on the conceptual model and screen design standards documents;

- Iterative detailed user interface evaluation – involves evaluating the previously un-assessed subsets of functionality, using a technique such as formal usability evaluation, iterating the process until all usability problems are addressed.

The final phase of the Usability Engineering Lifecycle is called Installation and involves only one task corresponding to user feedback. This task involves gathering feedback from users working with the product after it has been installed with the aim of resolving prevalent usability problems, enhancing the design or designing new releases or products.

There are also some general principles behind the Usability Engineering Lifecycle approach [Mayhew, 1999]. Notably, in the UEL user requirements and user interface drive the whole development process because the focus on interactive systems is to serve the users. Furthermore, the UEL fully adopts the idea that the integration of usability engineering with software engineering must be tailored in a way that enables different techniques to overlap and flow in parallel with development tasks. In addition, UEL fosters an iterative top-down structured process. The design/test/development phase is divided into three iterative levels where evaluation feeds-back specification and development. This characteristic is supported by a flexible and adaptable allocation of alternative usability methods and techniques, selected in conformance with the characteristics of the project. Finally, the UEL layers the development lifecycle across subsets of functionality, or in other terms, the development process progresses incrementally (see section II.4.1).

The UEL approach also provides guidelines for integration with object-oriented software engineering (OOSE) [Mayhew, 1999]. The integration is based on coupling OOSE models and workflows [Jacobson, 1992] with UEL tasks. Therefore, the user profile and contextual task analysis tasks should parallel the development of the requirements model in OOSE. The work re-engineering task corresponds roughly with the development of the analysis model. Finally, the actual user interface design, which starts with the conceptual model design task, enhances the design model. In the words of Mayhew this kind of integration is necessary because "(…) in OOSE, many fundamental Usability Engineering goals are still not addressed, and many fundamental Usability Engineering techniques are still not applied." [Mayhew, 1999].

II.5.2.User-centered Design

User-centered design is currently defined in the ISO 13407 Standard (Human Centered Design Processes for Interactive Systems). User-centered design (UCD) focuses specifically on the process required to build products usable with respect to the definition of usability described in section II.1.1. The user-centered approach typically entails involving users in the design and evaluation of the system so that feedback can be obtained. Therefore, UCD produces software products that are easier to understand and use; improves the quality of life of users by reducing stress and

improving satisfaction; and improves the productivity and operational efficiency of individual users and the overall organization [Daly-Jones et al., 1999].

II.5.2.1.Principles of User-centered Design

The ISO standard for human-centered design processes defines a set of principles that aim at incorporating the user perspective in the software development process. The principles are outlined as follows [ISO, 1999]:

- Appropriate allocation of function between user and system, determining which aspects of the job or task should be handled by people, and which can be handled by software and hardware. This division of labor should be based on the appreciation of human capabilities;
- Active involvement of users – utilize people who have real insight into the context in which an application will be used, therefore taking advantage of enhanced acceptance and commitment to the new software;
- Iteration of design solutions – entails the early and continuous feedback of end-users, through different prototyping techniques;
- Multi-disciplinary teams – fostering a collaborative development process which benefit from the active involvement of various parties, each of whom have insights and expertise to share.

II.5.2.2.Key Activities in User-centered Design

According to the ISO 13047 standard there are four essential user-centered design activities that should be undertaken in order to incorporate usability requirements into the software development process, as follows [ISO, 1999]:

- Understand and specify the context of use – the quality of the system, including user health and safety, highly depends upon the context in which the system will be used. Aspects that should be understood about the context of use include the characteristics of the intended users, the tasks the users will perform, and the environment in which the users will use the system;
- Specify the user and organizational requirements – the user-centered requirements should be explicitly stated alongside the technical and functional requirements. These include, identification of the range of relevant users and other personnel in the design, provision of a clear statement of design goals, an indication of appropriate priorities for the different requirements, provision of measurable benchmarks against which the emerging design can be tested, evidence of acceptance of the requirements by the stakeholders, acknowledgement of any statuary or legislative requirements;
- Produce designs and prototypes – to simulate the design solution(s) using paper or computer based prototypes from the earliest stages of design as well as during later stages. Prototyping enhances communication between the design team and

the end-users and provides an inexpensive way to test and explore design alternatives;

- Carry out user-based assessment – to confirm the extent to which user and organizational objectives have been met, as well as provide further information to refine the design. Evaluations should be carried out from the earliest opportunity and involve the following steps: develop an evaluation plan, collect and analyze data, report the results and recommendations for changes, iterate the activity until design (and usability) objectives are met, and finally tack changes, maintenance and follow-up.

II.5.3.Participatory Design

Participatory Design (PD) concerns the assessment, design, and development of technological and organizational systems that focus on the active involvement of potential or current users of the system in design and decision-making processes [Muller et al., 1993]. PD views users as the primary source of innovation, design ideas and information in the process of designing software intensive systems.

Participatory Design emerged in Europe, especially in the Scandinavian workplace democracy movement and also in England. The intent of PD was related to the importance of the social dimension of work, in particular, the concerns related to the social effects of new technologies in the workplace. From the initial Scandinavian and English traditions the field emerged in terms of number of practices, extent of theoretical development, number of practitioners and geographic dispersion. Today PD practice and research is diverse in perspective, background and areas of concern. Figure II.22 illustrates Muller's taxonomy of participatory design practices with respect to the position of the activities in the development cycle or iteration and the involvement of stakeholders in the activities [Muller et al., 1993].

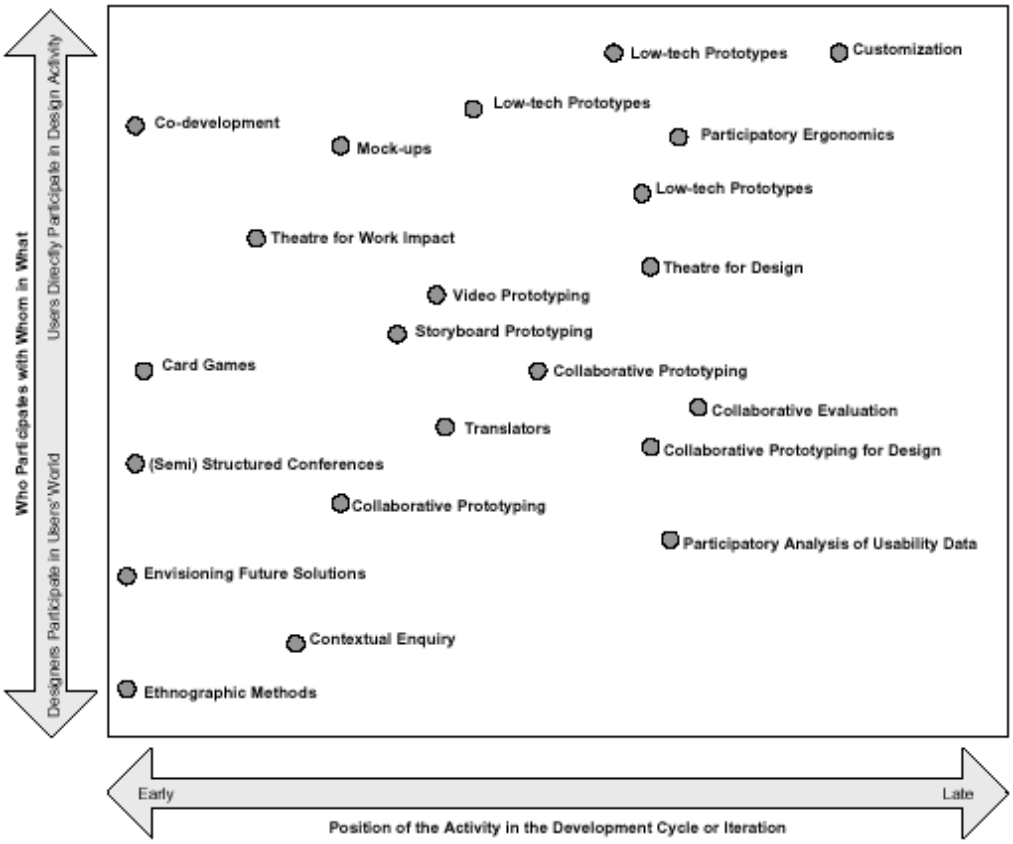


Figure II.22 – A Taxonomy of participatory design practices (adapted from [Muller et al., 1993]

Participatory techniques share a number of interrelated concerns as proposed by [Muller et al., 1995]:

- Representations – participatory techniques depend highly in the low-tech representations used to provide a concrete common language through which diverse stakeholders can articulate their views, clarify their disagreements, and work toward consensus;
- Group process – participatory techniques depend upon the social processes involved in the way the materials are used by a group of people – from egalitarian, collaborative approaches to arbitrary and even exploitive manipulations;
- Methodological rigor – participatory techniques should be specified with rigor and precision in order to enable integration with the software development processes.

Participatory design practices span the entire development lifecycle as we can see from the taxonomy in Figure II.22. Muller and colleagues map the different participatory techniques directly to the tools and practices involved in software development as follows [Muller et al., 1995]:

- Analysis – Participatory teams can manipulate low-tech representations of the people, objects, and tasks in their work domain to construct scenarios about existing and envisioned work. Other participants can manipulate, argue, query, propose and discuss alternatives about such representations;

- Design – Participatory teams can create and re-arrange low-tech representations to develop alternative scenarios for how work processes might be revised, extended and so on. These scenarios are a motivating basis for design and enable exploration of applications of different technologies;
- Assessment – Participatory teams can explore scenarios of how a particular scenario would impact real or envisioned work processes. This can be done relatively early in the software lifecycle, before the design is actually implemented.

II.6. ARCHITECTURES FOR INTERACTIVE SYSTEMS

Work on user interface architecture initially started with concerns about conceptual architectural models for user-interface objects (or entities). Examples of conceptual architectural models, depicted in Fig. 1, are MVC [Golberg and Robson, 1983], PAC [Coutaz, 1987] and the Lisboa Model [Duce et al., 1991]. Those models introduced the concepts of abstraction and concurrency to the user interface domain, through a set of collaboration agents (MVC and PAC) or objects (Lisboa). For a discussion and classification framework of conceptual architectural models refer to [Coutaz, 1993].

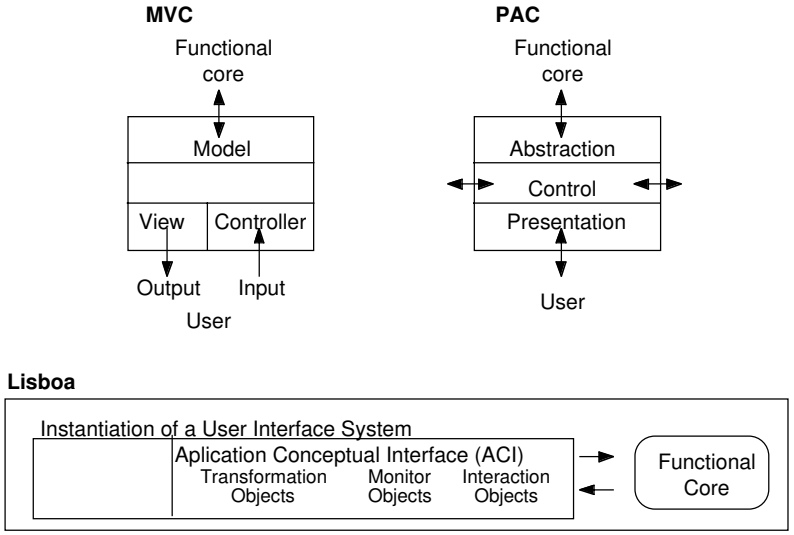


Figure II.23 - Conceptual Architectural Models for User-interface Objects

In the late 1980s and early 1990s several approaches of implementation architectural models for interactive systems emerged. Unlike conceptual models, implementation models organize the interactive system as a set of software components and aim at practical software engineering considerations like reusability, maintainability, and performance.

The Seeheim model [Bass, 1992] proposes a simple three-layer model (application, dialogue and presentation) of an interactive system, roughly coupling the semantic, syntactic and lexical functionality's of the user interface. The application component models the domain-specific components, the dialogue component defines the structure of the dialogue between the user and the application and, finally, the presentation component is responsible for the external to internal mapping of basic symbols. The Seeheim model is considered a correct and useful model for specification of interactive systems [Duce et al., 1991]. However, its support for distribution, concurrency, resource management and performance is recognized to be insufficient.

The Arch model [Pfaff and Haguen, 1985] is a revision of the Seeheim model aimed at providing a framework for understanding of the engineering tradeoffs, specifically, in what concerns evaluating candidate run-time architectures. The Arch model proposes a five-layer approach balanced around the dialogue component. The components of the Arch model are:

- The interaction toolkit component implements the physical interaction with the end-user;
- The presentation component provides a set of toolkit independent objects;
- The dialogue component is responsible for task-level sequencing, providing multiple view consistency, and mapping the between domain specific and user interface specific formalisms;
- The domain adapter component implements domain related tasks required but not available in the domain specific component;
- The domain specific component controls, manipulates and retrieves domain data and performs other domain related functions.

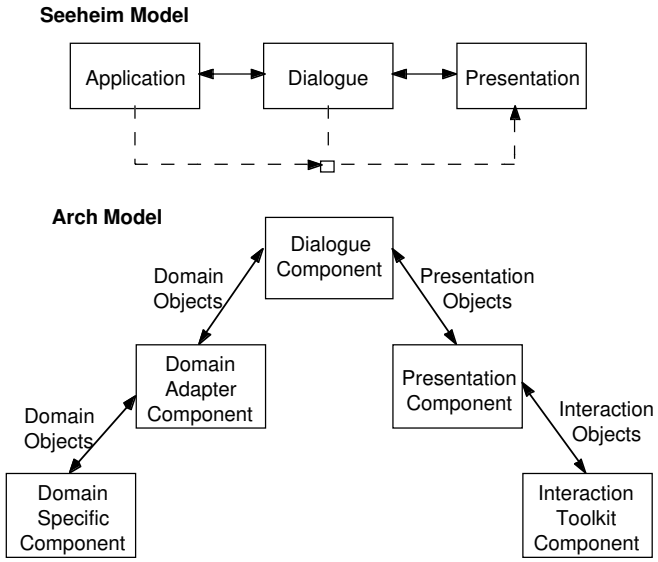


Figure II.24 - The Seeheim and Arch conceptual models

The Arch model provides developers with guidance to tackle the difficult engineering compromises that affect the development process of interactive systems and the quality of the end product. On the one hand the user and the functional core play a symmetric role driving the dialogue component, hence, at high levels of abstraction there is no a priori imposition on the control of the interaction [Coutaz, 1993]. On the other hand, both the domain adapter and interaction toolkit components serve as buffers for the functional core and the user, therefore, isolating and absorbing the effects of change in its direct neighbors [Coutaz, 1993].

The initial work on user interface architectures, in the late 80s and early 90s, suffered some refinements with the popularity of User Interface Management Systems (UIMS). Examples of more recent architectures supporting UIMSs are the Composite Object Architecture (for the Tube UIMS [Hill and Herrmann, 1987]), the ALV abstraction-

link-view architecture (ALV) (for the Rendevous UIMS [Hill, 1993]) and the Garnet architecture (for the Garnet UIMS [Myers et al., 1990]). These architectural models are usually referred to as concrete models, since they support actual implementations of UIMSs. However, they are closely related to the conceptual models described before, for instance PAC and MVC.

With the advent of the Internet, and the raising importance of distributed systems, the localization of the conceptual components can affect performance significantly. On a distributed system, like the Web, we have a number of places where computation can occur, the most obvious being the browser or web-client and the web server [Dix, 1999]. However, as the web evolves other places emerge where computation can take part, for instance proxy servers, database management systems, middleware servers, transaction servers and son on. The way an interactive application is split over these computational nodes affects resources and efficiency but also, and most important, usability considerations such as response time, feedback, security and so forth [Dix, 1999]. In collaborative systems there is the additional issue that users share views of the same data, the control over that data, and individual interaction. These issues inevitably lead to more complex decisions regarding user-interface architectures, both in terms of the conceptual and implementation models [Dix, 1999].

II.7. MODEL-BASED DEVELOPMENT OF INTERACTIVE SYSTEMS

Model-based development of interactive systems aims at identifying high-level models, which allow developers to specify and analyze the system under construction at a more semantic oriented level [Paternò, 2000]. Model-based systems exploit the idea of using a declarative interface model to drive the interface development process. An interface model is a declarative representation of all relevant aspects of a user interface, including the components and design of that interface [Puerta, 1997]. Interface models typically embody a number of interface related types at different levels of abstraction: user tasks, domain elements, presentations, dialogues, user types and design relations. These types are organized into different models within the overall interface model [Puerta, 1997]. That organization is expressed using an interface modeling language, which is a language that defines the different components and relationships of interface models.

The main advantages of model-based development approaches derive from the underlying declarative models (see section II.2.1). Declarative user-interface models provide the following advantages:

- They provide a more abstract description of the user-interface allowing designers to defer thinking about the low-level details, thus promoting exploration and innovation;
- They facilitated the development of methods that support systematic design and implement the user-interface;
- They provide the infrastructure required to automate tasks related to the process of user-interface design and implementation.

The work on model-based development of interactive systems emerged in the early 90s, in close relationships with the appearance of Model-based Interface Development Environments. Those systems enabled new possibilities, such as help generation, adaptive user interfaces, automatic user interface generation and multi-user interfaces over the common internal functionality. Concrete examples of model-based development environments (MBDE) have proliferated during the 90s, examples include ITS [Wiecha, 1990], UIDE [Foley et al., 1991], Humanoid [Szekely et al., 1992], L-CID [Puerta, 1993], Genius [Vanderdonckt and Bodart, 1993], Trident [Vanderdonckt and Bodart, 1993], Mastermind [Szekely, 1995], Adept [Wilson and Johnson, 1996], Fuse [Lonczewski and Schreiber, 1996], Tadeus [Stary et al., 1997], Tactics [Kovacevic, 1994], Mecano [Puerta, 1996], Mobi-D [Puerta, 1997]. An extensive review of those approaches is provided in [Silva and Paton, 2000].

II.7.1. Useful Models in Model-based Development Environments

Model-based approaches contrast implementation-oriented approaches, where the focus goes directly to implementation issues. Thus, model-based approaches benefit from a higher-level perspective that facilitates the management of the increasing complexity of interactive applications [Paternò, 2000]. This higher-level perspective is achieved through the use of models (see section II.2.1). The following is a discussion of the different models typically found in model-based approaches [Puerta, 1997]:

- Data models – Data models were the first explicit type of models to appear in model-based approaches, for instance UIDE used data models to produce static interface layouts. Data models enabled the generation of user-interface components (widgets) by matching data types with widget types. The main drawback of data models was that developers could not obtain a specification of interface behavior from them. Modern model-based approaches use data models as subcomponents of their interface model;
- Domain models – Following the advances in software engineering, model-based approaches started using entity-relationship data models (Genius), enhanced data models (Trident) and object-oriented data models (Fuse). Eventually, those models led to fully declarative domain models (Mecano), which effectively express the relationships among the object in a given context (see section III.3.3 for a more detailed discussion of domain models). Consequently domain models enabled partial specification of the dynamic behavior of the interfaces;
- Application models – although domain models were more elaborate than data models, they didn't describe the semantics of the application functional core and how they are related to the domain types. To address this problem approaches like UIDE, Trident and Humanoid introduced an application model in various forms. The aim of the application model was to facilitate the declaration of the interface behavior, achieved through actions and interaction techniques for which developers assigned parameters, preconditions and post conditions that enabled the control of the user-interface at runtime;
- Task models (or user-task models) – a task model describes the tasks that end-users perform and dictates what interaction capabilities must be designed. Task models are crucial in support of user-centered development (see section II.5.2). Task models involve elements such as goals, tasks and actions (see definitions in section II.1.2.2). In addition a task model can contain references to domain types that must be presented to the user to perform actions, tasks and goals. Task-oriented model-based approaches include Adept, Fuse, Tadeus and Trident, which use different forms of task models to drive the generation of alternative design solutions supporting the same interactive task;
- Dialogue models – the dialogue model describe the conversation between the human and the computer. It specifies when the end-user can invoke functions through various interaction techniques (buttons, command, and so on) and media

(voice, touch, gesture, and so on). It also specifies when the computer can query the user and present information. Dialogue models can be found in model-based approaches in various forms: dialog nets in Genius and Tadeus, attributed grammars in Boss, state-transition diagrams in Trident and dialogue templates in Humanoid;

- Presentation models – the presentation model specifies how interaction objects (e.g. widgets) appear in the different states of the dialogue model. Presentation models generally consist of a hierarchical decomposition of the possible screen displays into groups of interaction objects. Dialogue and presentation models are closely interrelated, thus they are typically considered together in model-based approaches (e.g. ITS).

All the models described previously are considered partial interface models because they don't convey all the relevant aspects of an interface design [Puerta, 1997]. The models described previously are those commonly found in model-based approaches, other models emerged over the years but never attained significant use – for instance platform, user and workplace models. Partial models have led to efforts in trying to define comprehensive interface models, such as those proposed in Mobi-D and Mastermind. As Puerta points out, "this evolution has followed that of component development itself. As interface models have evolved from data models to comprehensive model-based environments, there has been a corresponding move from elementary toolkit library components to basic textual or graphical editors, to prepackaged elements such as ActiveX that can be used to construct interfaces piecewise" [Puerta, 1997].

The confluence between comprehensive interface models and complex primitives (widgets and components) forms the basis of modern model-based approaches. Otherwise model-based approaches would require designers to work at a very low detail level.

In the following section we discuss task notations, one of the most important formalisms required to enable designers obtain this higher-level of abstraction.

II.7.2.Task Notations

As we saw previously task and dialogue models emerged from the need to complement domain and application models in order to convey the information required to: (i) describe the users tasks (task model) and describe the human-computer conversation (dialogue model). Task models are fundamental models for user-centered development because they enable designers to construct user-interfaces that reflect the real-world requirements of the end-users. In order to design the concrete user-interface the task model must be decomposed until we reach the basic tasks (or actions – see section II.1.2.2). When such a level of description is reached, the task model reflects the human-computer dialogue, thus the dialogue model. This

process, commonly known as the transition from task to dialogue [Bomsdorf and Szwillus, 1998], is fundamental for adequate user-interface design.

User-interface development approaches that rely on task modeling are commonly known as task-oriented approaches. Task-oriented approaches have received contributions from computer science, cognitive psychology and HCI (as illustrate in Figure II.25). In the cognitive psychology field the focus is related with the identification and characterization of human behavior (an activity known as task analysis). Conversely in the computer science field the focus is shifted towards finding notations suitable to represent tasks and their relationships to support interactive system development [Paternò, 2000].

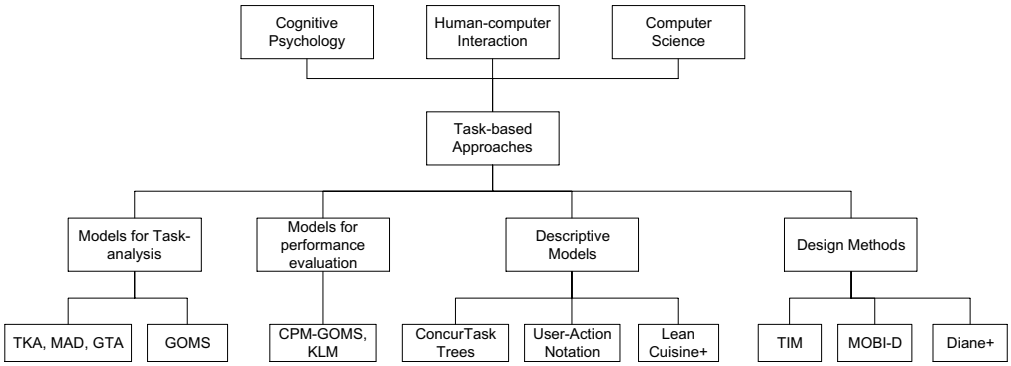


Figure II.25 – Different approaches to task models (adapted from [Paternò, 2000])

According to Paternò, task-oriented approaches can be classified in three different dimensions as follows [Paternò, 2000].

- The type of formalism used to represent the model – which varies from formal grammars to task-action grammars, production systems, knowledge models, transition networks and Petri nets;
- The type of information they contain – which can be oriented only to identify the activities and logical decomposition or also include indications of temporal relationships, artifacts manipulated and other properties;
- The type of support they provide – which can include evaluation, user-interface design, usability evaluation, and so on;

Many notations have been proposed for representing task and dialogue models. Generally those notations differ in the following aspects: the type of syntax (which can be textual or graphical), the level of formality (with respect to preciseness of the semantics), and the richness of operations that they offer to designers, which involves the capability to express sequential or concurrent operators [Paternò, 2000].

Initially task notations were proposed in the context of the model-based development environments they supported. Examples include hierarchical task analysis (Fuse), specialized Petri nets (Tadeus and Genius), attributed grammars (Boss), state transition diagrams (Trident), task knowledge structure (Adept), dialogue templates (Humanoid), and assorted notations with various levels of formality (Mecano, Mobi-D

and Mastermind). However, the most popular examples of diagrammatic task notation were state-transition diagrams (STDs), statecharts and petri nets. State-transition diagrams provide an easily displayed and manipulable representation based on a small number of constructs. However, they suffer from a number of shortcomings. Notably they are based on a "single event/single state" model, which can lead to an exponential growth in states and transitions in asynchronous dialogues, thus resulting in large and complex diagrams [Philips and Scogings, 2000]. Statecharts are basically an extension of STDs based on states developed to describe the behavior of complex reactive systems. Statecharts, which have been integrated in the UML (II.3), add a number of concepts to STDs (hierarchy, orthogonality and broadcast events) that provide increased expressive power at the cost of greater complexity. Although statecharts are an improvement over STDs, the need to explicitly show transitions continued to pose problems for task modeling, particularly for modern modeless interactive applications [Philips and Scogings, 2000]. Finally Petri-nets are abstract virtual machines, with well-defined behavior that have been mainly used to specify process synchrony in real-time systems. Like STDs, Petri-nets define possible sequences of events, but also require transitions to be explicitly shown, thus retracting and cluttering the diagrams.

The problems with the previous notations and the limitations of model-based development environments, which lead to their failure in reaching industrial application [Myers et al., 2000], forced a new perspective for task notations. From an initial focus on their capability to drive the automatic generation process (motivated by the purpose of model-based environments), task notations started reflecting the requirements of the user-interface designers. Thus, the second generation of task notations is clearly more focused on the usability and expressive power of the notation to facilitate user-centered development and user-interface design. This change in the field is evident in the words of Puerta: "Instead of automation, we looked for ways to support interface designers. Instead of large knowledge bases and complex inferencing strategies to make design decisions, we built interactive tools for developers to do what they do best: design interfaces. Instead of using interface models as a driving mechanism for an expert system, we used them to organize and visualize interface design knowledge" [Puerta, 1998].

Examples of these second-generation task notations are LeanCuisine+ [Philips, 1995], Diane+ [Tarby and Barthet, 1996] and ConcurTaskTrees [Paternò, 2000] notations. In the following we summarize the main characteristics of these notations:

- LeanCuisine+ is a semi-formal graphical notation for describing the behavior of event-based direct manipulation graphical user-interfaces [Philips, 1995]. In a LeanCuisine+ specification the interface is represented primarily as a dialogue tree. The behavior of the interface is expressed in terms of constraints and dependencies, which exist between the nodes of the tree (called menenes). The major advantage of LeanCuisine+ is that task action sequences can be represented

in the context of the structure of the interface dialogue. Although LeanCuisine+ was not developed in the context of a model-based development environment, there are reports of automatically generating LeanCuisine+ diagrams from prototype code. Also, the notation still lacks the support of a modeling tool;

- Diane+ [Tarby and Barthet, 1996] is actually a design method that embodies a diagrammatic task-based notation. In the Diane+ notation corresponds to a multi-level structure of tasks, where the highest level corresponds to procedures associated with goals. Diane+ procedures are refined through tasks, sequenced with precedences that express chronological chaining, coupled where required with constraints. It is partially possible to generate and manage the user interface and contextual help directly through Diane+. Moreover, a modeling tool supporting Diane+ and some automation mechanisms is described in [Lu et al., 1998];
- ConcurTaskTrees (CTT) [Paternò, 2000] is a notation developed taking into account the previous experience in model-based approaches and adding new features in order to obtain an easy-to-use and powerful notation. CTT is a graphical notation where tasks are hierarchically structured and combined using a set of temporal operators (see examples in section IV.4.4). The formal semantics of CTT have been defined [Paternò, 2000] and a modeling tool supporting the development of CTT models is publicly available (CTT Environment - CTTe). Moreover, CTT is considered one of the most widely used task notations in the usability-engineering field.

Despite the importance of task notations for user-centered development and user-interface design, and the relative success of 2nd generation notation such as CTT, there is no consensus over a task-modeling notation in the usability-engineering field [Puerta, 1997]. The UML provides two types of notations that can be used to perform task or dialogue modeling. UML use-case diagrams to some extent can be used to perform user-task modeling. However, there is consensus that use-cases pose many problems has a general notation for modeling the users tasks. Section III.3.1 discusses those problems in detail. The UML also includes statecharts (UML statechart diagrams), so there is the possibility of performing dialogue modeling as proposed in early model-based development environments. However, has we discussed previously, statecharts are widely recognized to be hard to use for complex interactive applications.

There is an important benefit in integrating task notations with object-oriented notations, in particular the UML. Such integration would leverage the communication between the models describing the application internals and the models describing the user tasks and the user-interface. Moreover, since the UML benefits from an increasing tool support and interoperability, task notations could benefit from those facilities. Different attempts have been carried out in this direction. For instance conjectures about an integration of LeanCuisine+ with the UML is presented in

[Philips and Scogings, 2000], and a description of a pragmatic transformation between Diane+ and UML state diagrams is described in [Lu et al., 1998]. An adaptation of CTT to the UML was proposed in [Nunes and Cunha, 2000b] and described in section IV.4.4.

II.8. CONCLUSION

This chapter presented a survey of usability engineering and object-oriented software engineering bringing into context the two major research fields addressed in this thesis.

The main conclusion we can draw from this survey is the insufficient integration of usability engineering in modern software development practice. Although, usability engineering has matured as an engineering discipline in the past decade, it is irrefutable that an appropriate integration with modern software development processes was not accomplished.

The different usability engineering models and techniques presented in this chapter are established and enable an effective engineering approach to usability. The attributes of usability are well known and can be effectively measured, evaluated and improved. The existing cognitive frameworks of usability enable a precise, quantitative and qualitative, estimation and evaluation of human-performance. Decades of experience designing interactive systems are documented in extensive lists of design heuristics. The main tasks forming the usability engineering lifecycle are identified and provide a broad framework for introducing different HCI techniques. Finally, a standard for human-centered design process, describing the principles and key-activities for effective interactive system development, was approved recently.

Meanwhile, the software engineering community achieved a *de jure* and *de facto* standard for object modeling whose success is unquestionable. The Unified Modeling Language (UML) is recognized to be the dominant diagrammatic modeling language in the software industry. It has found widespread use in many different application domains, from avionics and real-time systems to e-commerce and computer games. It applies to the different modern requirements to build and document software intensive systems. It is used for business process modeling, to describe software architectures and patterns, and maps to many different middleware architectures, frameworks and programming languages. The UML is also recognized to have influenced the CASE tool industry, improving the quality and availability of high-end case tools, in particular for modeling purposes.

Although the UML standard doesn't provide a methodological background, the Unified Process, and its' variations, define a modern iterative and incremental, architecture-centric and use-case driven software development process, highly integrated with the UML. Those approaches recognize the importance of user acceptance and involvement as a key aspect of modern software development.

Multiple references to *real value to the users*, *user involvement*, *real-world tasks*, *business critical use-cases*, are common in UML based processes. However, the importance of user-centered design and user-interface design is scarce. Constantine, in a keynote speech to the TOOLs'2000 conference [Constantine, 2000], quantified the impact of user-interface design in terms of the number of pages in both the UP and RUP textbooks. The RUP textbook includes 4,5 pages (out of 262) about user interface design and the UP 12 pages (out of 450) with a sidebar on essential use-cases. Although this quantification is arguable, it is irrefutable that user-interface design is not given an adequate importance in modern object-oriented development. In the same keynote speech Constantine reinforced his point highlighting some quotes of the mentioned textbooks, as follows:

- "We care about user interfaces… only if they are architecturally interesting. However, this is seldom the case." [Kruchten, 1998]
- "Based on feedback, we massage the prototype until is satisfies the needs of users… The prototype them becomes the user interface specification" [Kruchten, 1998]
- "Designing the user interface is part of the requirements workflow and not design." [Jacobson et al., 1999]

Despite the lack of integration between usability engineering and software engineering, some of the problems that both disciplines face are highly interconnected. Software engineering achieved impressive results in a very limited amount of time - only 50 years have passed since the invention of the modern computer and already we're faced with the ubiquity of computing. Although at the first time we can consider that usability engineering has been marginal to the development of software engineering, when we look closely to what has been accomplished we find many intersection points. Some of the most important achievements in software engineering are related to research in usability engineering - irrefutable examples are object-oriented programming, component-based development, event languages and the current desktop graphical user-interface [Myers, 1995]. Although usability specialists have conventionally been considered *outsiders* in software development, their role has increased significantly in the past years. From the early focus on product assurance testing (human factor specialist) they have been evolving to more integrated activities such as integrated product design and development (usability specialist and user-centered specialists) [Karat, 1997].

More evidence of this close relationship can be found in research challenges that software engineering faces today. In a recent book about the future of software engineering, which celebrated the last international conference on software engineering of the century (ICSE'2000) [Finkelstein and Kramer, 2000], the authors identified several research challenges. Four out of eight major research challenges are

very familiar to usability engineering. To illustrate this point we reproduce of those questions in the following [Finkelstein and Kramer, 2000]:

- "Change - How can we cope with requirements change? How can we build systems that are more resilient or adaptative under change? How can we predict the effect of such changes"
- "Service-view – How can we shift from a traditional product-oriented view of software development towards a service view? What effects do new modes of software service delivery have on software development?"
- "Non-classical life cycles – How can we adapt conventional software engineering methods and techniques to work in evolutionary, rapid, extreme and other non-classical styles of software development?"
- "Configurability – How can we allow users, in the broadest sense, to use components in order to configure, customize and evolve systems?"

To different extents all of those issues are related to the fact that software intensive systems are driven and used by people. For instance, the problem of requirements change and configurability is a well-known issue in usability engineering [Myers, 1994; Hackos and Redish, 1998; Beyer and Holtzblatt, 1998]. Similarly, techniques supporting non-classic lifecycles - for instance participatory techniques [Muller et al., 1993], prototyping [Isensee and Rudd, 1996] and the role of creativity in design [Laurel and Mountford, 1990] – have been extensively studied in the usability-engineering field. Finally the shift from a product to a service view is to some extend motivated by the advent of the increase connectivity of computing [Baecker, 1992] and information appliances [Norman, 1998]. Obviously, usability engineering is no silver bullet to those problems [Myers, 1994], but much progress can emerge from interdisciplinary work.

Conversely, some of the problems that usability engineering faces today can obviously benefit from previous research in software engineering. The increasing complexity of modern user-interfaces poses difficult engineering problems that can only be solved taking advantage of the recent advancements in software engineering (for instance in the domains of software architecture, software infrastructure, and so on). In particular, the maturity of object-orientation and the advent of the UML and associated tools can highly benefit a major problem in usability engineering: the lack of a non-proprietary standard language that leverages tool support and interoperability. In the following chapter we discuss the state-of-the-art in this topic of object modeling for user-centered development and user-interface design.

III. State of the Art: Object Modeling for User-Centered Development and User Interface Design

"To steal ideas from one person is plagiarism; to steal from many is research."
—Unknown

This chapter describes recent developments in the field of object modeling for user-centered development and user interface design. The aim of this chapter is to review and discuss recent work related to the main topic of the thesis, therefore justifying that the research problem is previously unsolved. The next chapter will build on the definitions, discussions and arguments, presented here, to explain why Wisdom solves the problem of building usable software intensive systems under the framework of the UML.

The chapter starts with a brief historical perspective of object modeling and user interface design. From the initial efforts of achieving usability through consistency and standards in the 1970s and 1980s to the results of the latest workshops on the subject of object models for user-interface design held at major HCI and SE conferences from 1997 to 2000.

The second section of this chapter extends the discussion of the previous section with a specific focus on the recent developments in object modeling for user-centered development and user interface design (ranging roughly from 1995 to early 2001). This review of the state-of-the-art starts presenting different view models for user-

interfaces and relates them with the existing view models of the UML. A conceptual framework, developed at the first workshop discussing object models and user interface design, presents a unifying view model of the different dimensions required to bridge the gap between HCI and SE. The dimensions are isolated as process, notation (and semantics) and architecture.

The third section of this chapter presents the useful models identified in the conceptual framework and required to adequately support user interface design with object models. We present and discuss the different models in the framework (task, business process, domain, interaction, user-interface and interactive system models), contrasting definitions from the various fields involved in the development of interactive software intensive systems: requirements engineering, human-computer interaction and OO software engineering.

The fourth section takes the previously defined unifying framework and presents the recent proposal for integrated user-centered OO methods. We clearly define what characterizes those new methods and illustrate how they resolve the problems encountered in conventional OO development methods. We then review three UC-OO methods that emerged in the late 90s: Ovid, Idiom, and Usage-centered Design. The section and the chapter end with a description of an encompassing UC-OO process framework used to illustrate the requirements of an ideal solution for the problem of bridging HCI and SE with an OO method that takes the best from both fields and produces highly usable software intensive products.

III.1. BRIEF HISTORICAL PERSPECTIVE OF OBJECT MODELING FOR USER-CENTERED DEVELOPMENT AND USER INTERFACE DESIGN

Until very recently both research and practice didn't focus on the application of object orientation to the overall organization and coordination of the different processes, models, artifacts, behavioral and presentation aspects of interactive systems. Through the 1970's and 1980's there were many efforts to achieve usability through standardization and consistency. Such principles ultimately gained visibility with the Xerox "Star" and later the Apple Macintosh, which provided the first commercial products deploying graphical user interfaces (GUIs) based on the direct manipulation interface style. Direct manipulation refers to systems that make visible the objects of interest to users enabling rapid, incremental and reversible actions on those objects by direct manipulation instead of complex command language syntax [Shneiderman, 1998]. Direct manipulation is usually confused with the term graphical user interface (GUI), which refers to a user interface technology that requires bitmapped displays, as opposed to character-based displays. Hence, direct manipulation describes a style of interaction, while GUI refers to a specific interface technology.

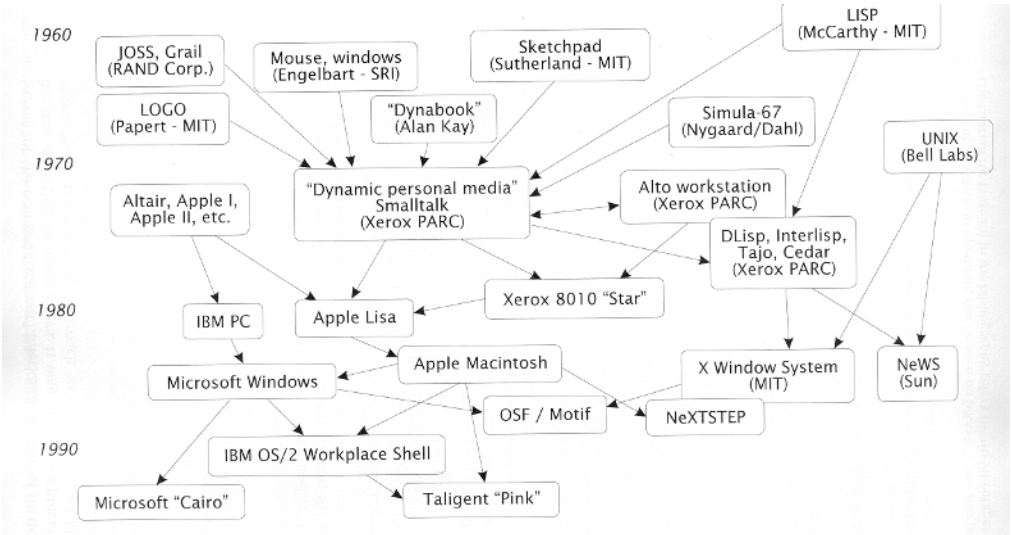


Figure III.1 - The history of OOUI by Collins [Collins, 1995]

During the 80s, others pioneered the use of object technology describing architectures for user-interface objects (e.g. MVC and PAC) and logical input devices [Golberg and Robson, 1983; Coutaz, 1987; Foley et al., 1990] (see also section II.6). However, only in the late 1980s, the concept of an object-oriented user interface was explored by IBM's CUA architecture, and incorporated in IBM and Microsoft products [IBM, 1991].

One of the first explicit references to object-oriented user interface design was provided by Collins [Collins, 1995], describing a principled approach that applied object-orientation to end user-interfaces and the internal system structure that implements them. Collins elaborated the MVC implementation architecture that corresponds to a three-way factoring of user interfaces into conceptual model, presentation language and action language.

But it was not until the 1990's that the first attempts to use object technology to support analysis and design of interactive systems emerged. Jacobson and colleagues introduced the concept of use-cases and defined the entity, interface (boundary in the UML) and control analysis framework, which in part accomplished the separation of concerns between internal functionality and the user interface. In his work [Constantine, 1992] extended use-cases in order to enable the connection between the structure of use and the structure of the user interface. The three models (user role model, use-case model and context model) of Constantine's essential use-cases provided a way to connect the design of the user interface back to the essential purpose of the system and the work it supports. Since Constantine's proposal use-cases have seen a plethora of definitions and extensions, contributing to its ubiquity in software development, but also compromising its utility (see [Hurlbut, 1998] for an extensive review of different approaches for describing and formalizing use-cases).

Different authors addressed the representation of conceptual models using object technology. Examples include the following methods: STUDIO [Browne, 1993], Idiom [Harmelen, 1996], GUIDE [Redmond-Pyle and Moore, 1995] and the LUCID approach [Kreitzberg, 1996]. A notable exception to the use of object-modeling was ERMIA that used entity-relationship diagrams to express conceptual and perceptual models for user interface design [Green, 1991; Benyon and Green, 1995]; [Green and Benyon, 1996]. Despite the data modeling nature of ERMIA, the approach addresses common concerns with its object-oriented counterparts. ERMIA defines conceptual descriptions of information artifacts (conceptual models) and perceptual descriptions, which convey access to the conceptual model and employ a method of presentation from which users may derive information [Sutcliffe and Benyon, 1998].

Also during the late 90s, Roberts and colleagues [Roberts et al., 1998], in the Ovid method, introduced the concept of an object view and its relationship to user perceivable objects. Constantine and Lockwood also proposed an object-oriented based approach for interactive systems design [Constantine and Lockwood, 1999]. Their Usage-centered Design approach expands early work on essential use-cases, defining a set of essential models for usability: a role model (describing users and user roles), a task model (describing users' work and tasks) and a content model (describing interface contents and navigation). Despite, to some extent, enabling the use of the UML as their notational basis, both Ovid and Usage-centered Design, are not fully compatible with the UML standard.

Between 1997 and 2000, several workshops discussed the role of object models and task/process analysis in user interface design [Harmelen et al., 1997; Artim et al., 1998; Nunes et al., 1999; Nunes et al., 2000]. All this work, emerging both from industry and academia, stressed the importance and the opportunity of using object-technology to bridge the gap between practice in software engineering and HCI. The framework develop during the CHI'97 Workshop [Harmelen et al., 1997] at the Computer Human Interaction (CHI) Conference, was consistently used to depict the problem of using object models to bridge the gap between SE and HCI. This framework was refined by Kovacevic [Kovacevic, 1998] and subsequently used in the ECOOP'99 workshop (WISDOM'99) [Nunes et al., 1999] to identify the dimensions that need to be considered to effectively bridge the gap between HCI and SE using object models.

The following CHI'98 workshop concentrated on discussing the problem of incorporating work, process and task analysis into commercial OO software development [Artim et al., 1998]. The workshop resulted in a set of extensions to the UML metamodel that aimed at enabling HCI work in close collaboration with object-oriented development. However, this extension had several problems (as discussed in section III.3) and failed the purpose of disseminating to the SE community.

The goal of disseminating some of the concerns addressed in the CHI workshops to the SE community was accomplished in the ECOOP'99 workshop held in 1999 at the European Conference in Object-Oriented Programming (ECOOP'99) [Nunes et al., 1999][1]. During this workshop, the participants (from the HCI and SE communities) reviewed and discussed the outcome of the previous CHI workshops. The workshop succeeded in stressing the importance of using object-technology to integrate SE and HCI practice. A subset of the revised CHI'97 framework, originally proposed by Kovacevic [Kovacevic, 1998], was considered and four important dimensions discussed: process, notation, architecture and traceability. Furthermore, an "ideal" integrated User-centered Object-Oriented (UC-OO) process framework was discussed and proposed.

From the community involved in the different workshops, emerged an edited book proposal that included some of the most important approaches developed during the 90s, including integrated UC-OO methods, OO based techniques for user-interface design and reviews and discussions about conventional OO approaches to user-interface design [Harmelen, 2001a].

All this work urged the different contributors in the field to approach the UML community and propose a UML profile for interactive system development. The discussion of that proposal was initiated in the Tupis'00 workshop [Nunes et al., 2000], held in 2000 at the International Conference on the Unified Modeling Language.

[1] The Workshop on Interactive System Design and Object Models (WISDOM'99), held at the European Conference on Object Oriented Programming (ECOOP'99), is hereinafter designated ECOOP'99 Workshop to avoid confusion with the Wisdom method (Whitewater Interactive System Development with Object Models).

During the Tupis'00 workshop the participants discussed the requirements for an UML extension for interactive system development. There was consensus that such extensions required mechanisms that are not currently supported in UML 1.4 standard. However, that problem was shared with other parties involved in different extensions and is actually one of the greatest requirements for the next major UML revision (UML 2.0). The fact that the new UML version is moving towards a family of languages with a common, smaller and semantically well-defined core was considered appropriate to accommodate the envisioned extensions for interactive system development.

III.2. RECENT DEVELOPMENTS IN OBJECT MODELING FOR USER-CENTERED DEVELOPMENT AND USER INTERFACE DESIGN

The previous section briefly summarizes some early developments in using object technology to support user-centered development and user-interface design. Most of the work until the late 90s concentrated on using object technology to represent conceptual and perceptual models of user interfaces. This work emerged mainly from the human computer interaction (HCI) community, which also contributed with proposals for conceptual and implementation architectures for interactive systems (summarized in section II.6).

From the perspective of the software engineering community, the most important contribution is usually recognized to be Jacobson's concept of use-cases. However, Martin and Oddel argue that use-cases are actually operations associated with triggers and events that correspond to interfaces expressed in terms of stimulus and responses [Martin and Odell, 1998]. This approach, originally described by McMenamin and Palmer [McMenamin and Palmer, 1984], is used to perform context specification, which is defined as the expression of relationships between a given system and its environment [Martin and Odell, 1998]. Context specifications appear in context diagrams that convey a high-level perspective of the system, enabling the need to specify all the various structural and behavioral elements of a system on a single model.

In following sub-sections we present the recent developments in using object modeling for user interface development. This presentation, not only describes different approaches, but also provides a critical discussion of the topics related to the major contributions of the thesis. Therefore, the justification that the problem addressed in this thesis was not solved previously is given in this section.

III.2.1.Three Views of User Interfaces

In [Sutcliffe and Benyon, 1998] the authors propose a high level perspective of model-based design of user interfaces corresponding to a three level view of user interfaces. The three views are as follows [Sutcliffe and Benyon, 1998]:

- Structural view – focuses on the main entities or objects, which are in the system and how those object are related. The two main traditions of structural models in software development are entity-based and object-oriented. Thus, structural models are usually expressed as object models and entity-relationship models.

- Functional view – focuses on how some substance or object moves through the system. Functional aspects include data flows between processes (workflows), models of the control flow and movement of physical objects (sequencing of actions) or the physical actions that people have to undertake and the things they have to think about and remember. Functional models of interactive systems are usually expressed with task models or use-cases models.
- Dynamic view – focuses on how the structure and functioning of the system are related and the behavior that results. The model of system dynamics concentrates on the events that occur and the different states that the system can take up depending on those events. Dynamic models of interactive systems are usually expressed with statecharts.

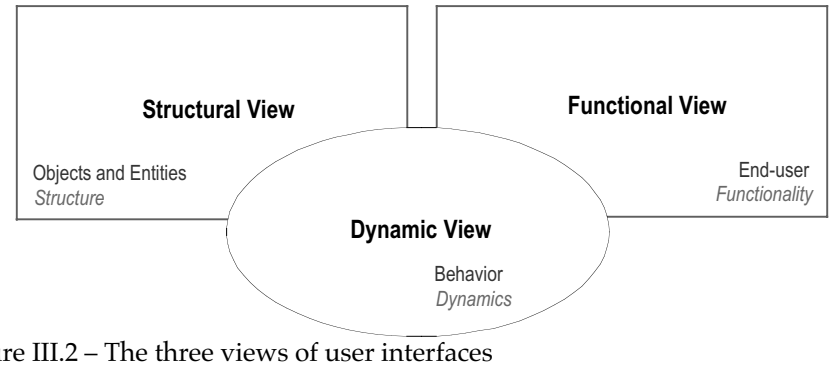


Figure III.2 – The three views of user interfaces

An interesting aspect of the three views of user interface, with respect to the object modeling approach, is trying to find a clear mapping between the user-interface views and the views proposed in the different interpretations of the object modeling approach. The UML standard defines a view as "projection of a model, which is seen from a given perspective and omits entities that are not relevant to that perspective" [OMG, 1999]. However, the standard itself doesn't define specific views, that is, it omits any reference to structural, dynamic or any other kind of view. This is consistent with the ideas of Rumbaugh and colleagues that point out that a view "is simply a subset of UML modeling constructs that represents one aspect of a system" [Rumbaugh et al., 1999]. Thus, views are used arbitrarily to organize concepts and notational elements in ways that are convenient and intuitive for modelers [Rumbaugh et al., 1999].

Therefore we can find in the literature different views defined for different purposes. One example is provided in section II.4.3.1 with the logical, implementation, process, deployment and use-case view of the 4+1 model view of architecture. Another one is given in [Rumbaugh et al., 1999] with the structural, dynamic, model management and extensibility views. Finally the Unified Process promotes yet another approach that encompasses the analysis, design, deployment, implementation and use-case views. They are in fact referred to as models, but are quite consistent with the notion of a view (as defined before) if we assume that models are at a given point in time

views until they convey a complete description from the same given perspective (see section II.4.3.1).

View models also appear, in a different form, in the foundation of object-oriented modeling described in section II.2.4. Martin and Oddel divide the basic level of their foundation terms of structural and behavioral specifications, which map consistently with both the three views of user-interface and two of the views proposed by Rumbaugh and colleagues. However, Martin and Oddel foundation differs from Rumbaugh's approach since it considers use-cases as being part of "other modeling approaches", in particular context specification.

III.2.2.A Unifying Framework for Object Modeling for User Interface Design

The ECOOP'99 workshop version of the CHI'97 framework can also be conceived as a view model of interactive systems, comprising different perspective of interest for interactive system development. Figure III.3 illustrates the Wisdom'99 revised framework of the CHI'97 workshop, originally proposed in [Kovacevic, 1998].

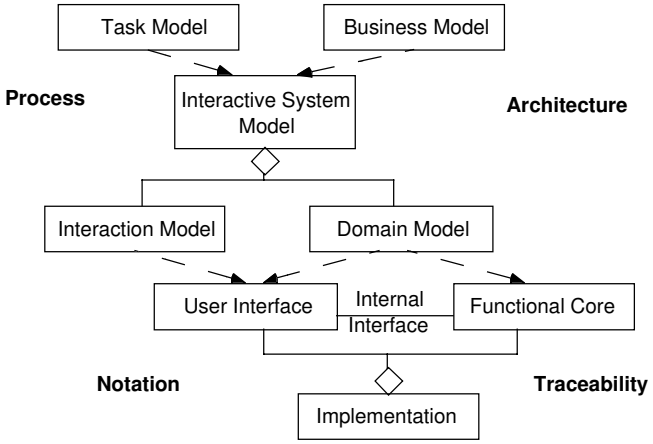


Figure III.3 - The ECOOP'99 version of the CHI'97 framework, including the three issues – notation, process and architecture [Nunes et al., 1999; Kovacevic, 1998; Harmelen et al., 1997]

This view model architecture clearly (logically, not necessarily physically) separates the functional core from the user interface. While the user interface depends on both the domain and interaction models, the functional core only depends on the domain model. This view model also supports the separation of the presentation (in the user interface) from the conceptual specification of the interaction (the interaction model).

The framework in Figure III.3 passed extensive scrutiny in the CHI'97, CHI'98 and ECOOP'99 workshops. Kovacevic independently defined a set of dimensions for integrating user-interface design into software development [Kovacevic, 1998]. Those dimensions were used and explored in the ECOOP'99 workshop and are described in the following section. In addition, a set of definitions for the different models in the framework where consensually defined in the ECOOP'99 workshop and included

contributions from both the object-oriented software engineering (OOSE) and the human-computer interaction (HCI) communities.

III.2.3. Dimensions for Integrating User-Interface Design into Software Development

The three dimensions for integrating user-interface design into the software development lifecycle, as illustrated in Figure III.3, are as follows [Kovacevic, 1998]:

- Notation and Semantics - The same argument that drove adoption of the UML – to enable tool interoperability at semantic level and provide common language for specifying, visualizing and documenting software intensive systems – applies to bridging the gap between user-interface design and OO software engineering. Both UI and OO developers should collaborate in developing the domain model, although focusing on different aspects. In addition, OO analysis and design focuses on refining the domain model toward an implementation of the functional core. The domain model captures the semantics of the context of the system and defines the information requirements for the application UI. User interface design focuses on the interactive model that contains the UI specific information complementing the domain model. Together the domain and interaction models define a model of an interactive system. Notation must facilitate this collaboration by supporting any additional UI related modeling views, and providing the underlying semantics that tie UI design constructs with OO development constructs;
- Architecture – Interactive systems require an architecture that maximizes and leverages user-interface domain knowledge and reuse. As mentioned in the previous section, the model in Figure III.3 separates the functional core from the user-interface, including the presentations aspects and abstract interaction model. Therefore this model leverages on user-interface domain knowledge by providing design assistance (evaluation and exploration) and run-time services (UI management and context-sensitive help).
- Process – As mentioned in section II.5.1, usability and consequently user-interface design must be incorporated into a software development process as an integral part, not an afterthought. Only redesigning the whole development process around usability engineering requirements we can facilitate collaboration between usability experts, user-interface designers and software developers.

The approach proposed by Kovacevic leverages a set of benefits that an envisioned user-interface design tools could support. The benefits described in [Kovacevic, 1998] are as follows:

- Evaluation - Different UI designs can be evaluated with respect to specific tasks and/or given criteria, such as speed and frequency of use, learning time, and associated costs [John and Kieras, 1996b; Kieras et al., 1997; Sears, 1995];

- User-interface generation - A user-interface implementation can be directly produced from a model according to the model-based tradition [Johnson et al., 1995; Kovacevic, 1994; Szekely et al., 1996];
- Help generation - Context-sensitive help, as a run time component of the application user-interface, can be produced including animations and explanations [Foley et al., 1991; Sukaviriya and Foley, 1990];
- Consistency/completeness checks - A user-interface can be evaluated with respect to completeness and consistency, for instance checking if all user actions can be reached or enabled [Kovacevic, 1994];
- Design assistance in exploring the user-interface design space [Foley et al., 1991; Kovacevic, 1994; Szekely et al., 1992].

III.3. USEFUL MODELS IN OBJECT MODELING FOR USER-CENTERED DEVELOPMENT AND USER INTERFACE DESIGN

In the following sub-sections we use the ECOOP'99 workshop version of the CHI'97 framework to present, define, contrast and discuss different interpretations of the models encompassing that framework (depict in Figure III.3). The presentation aims at reviewing the recent developments in the field in an integrated way and providing references to related work in both SE and HCI fields. In addition we discuss the potential of object modeling to convey the required information in each model.

III.3.1.Task Model

Task modeling is a central and familiar concept in HCI but seldom-used in object-oriented software engineering. A commonly accepted definition of a task model, from the HCI perspective, emerged in the CHI'97 workshop: "task models describe the activities in which users engage to achieve some goal (...) A task model details the users' goals and the strategies adopted to achieve those goals, in terms of the actions that users perform, the objects involved in those actions and the sequencing of activities" [Harmelen et al., 1997]. This definition is consistent with the definitions provided by the model based user interface design approach (see section II.7).

From an OOSE perspective, task models can be considered similar to use-case models. However the definition of a use-case model only refers to system functionality that provides users with results of value (see section II.4.1).

As we discussed in [Nunes and Cunha, 2000b], the adoption of use-cases in the UML acknowledges the importance of identifying the different roles users play when interacting with an application to support their tasks. However they are still mainly used to structure the application internals and don't provide an efficient way to support the usability aspects of the interactive system. Essential use-cases [Constantine, 1992] identified some of the requirements for user interface development. An essential use-case is "a simplified, abstract, generalized use-case defined in terms of user intentions and system responsibilities" [Constantine, 1992], it contrasts Jacobson's original definition of a concrete use-case "a class describing the common pattern of interaction of a set of scenarios" [Jacobson, 1992].

Essential use-cases define user role maps, a set of relationships between actors (affinity, classification and composition) that reveal how the different roles in a system

fit together defining who will use the system and how [Constantine, 1992]. Other approaches, used in goal-driven interaction design, rely on archetypal descriptions of users. In one of those proposals [Cooper, 1999] the author defines a persona, a hypothetical archetype of a user defined with precision and detail. The author claims that instead of designing for a broad audience of users, interaction designers should design for a small set of archetypal users – the cast of characters [Cooper, 1999]. The requirements in such proposals are substantially different from the common interpretation of use-cases in OO analysis and design – they recognize the peculiarities of interaction design and they focus on users' goals and the structure of use. A proposal for structuring use-cases with goals [Cockburn, 1997] already recognizes some of the above problems distinguishing goals as a key element of use-cases.

The descriptions of user behavior are intimately associated with the technique (and underlying concepts) used to abstract the end-users. In the UML, and associated development methods, descriptions of user behavior are usually captured detailing use-cases with scenarios, activity diagrams or interaction diagrams. From an HCI perspective the descriptions of user behavior usually encompass a task model, which, at least to some degree, can be achieved with the mentioned UML diagrams – in [Artim et al., 1998] and [Kovacevic, 1998] two UML extensions are proposed to accommodate task models.

However the major problem with descriptions of user behavior are related to the system-centric nature of use-cases. In [Constantine and Lockwood, 2001] the authors argue that "conventional use-cases typically contain too many built-in assumptions, often hidden and implicit, about the form of the user interface that is yet to be designed." Such argument led the authors to propose the essential use-case narrative, a technology-free, implementation independent, structured scenario expressed in the language of the application domain and end-users. The essential quality (technology free and implementation independent) of descriptions of user behavior can also be applied to diagrammatic representations. In [Nunes and Cunha, 1999] we propose to detail use-cases with activity diagrams following the same principle. Goal-driven interaction design also relies on descriptions of user behavior structured in terms of users' goals. For instance in [Cooper, 1999], the author proposes to drive design in terms of personal, corporate and practical goals.

This view of the role of task modeling, with respect to conventional use-case modeling, was also stressed in the results of the CHI'98 workshop. Artim and colleagues pointed out "the difficulty with UML's approach to use cases is that it is not a generalized form of task modeling. A number of important features such as decomposability and task frequency are entirely missing from the UML model" [Artim et al., 1998]. As a result of this problem, the workshops participants proposed an extension to the UML metamodel (depict in Figure III.4) that defined a separate

user task model and provided a standard model for traceability between tasks and their corresponding use-cases.

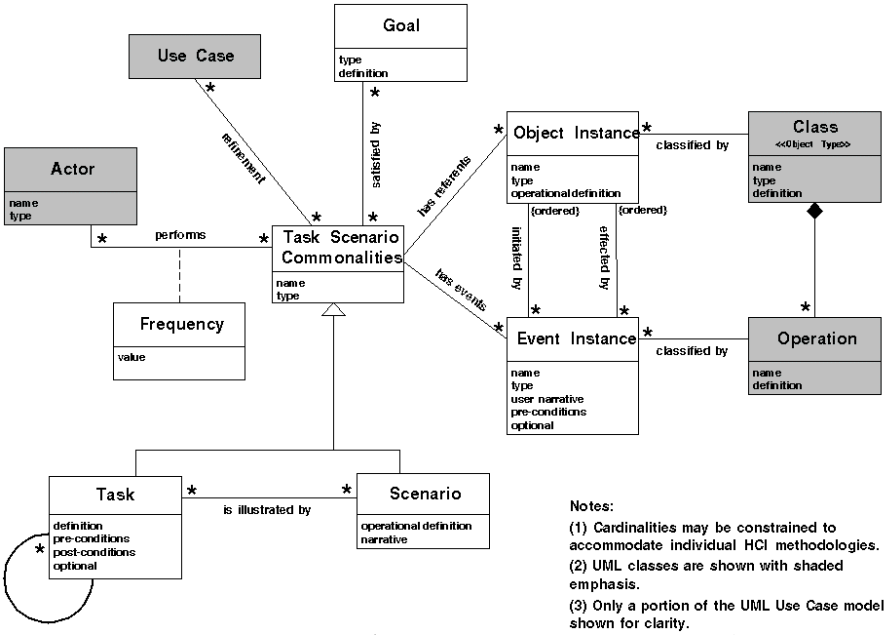


Figure III.4 – Original draft of the CHI'98 User Task Model (from [Artim et al., 1998])

The problem with the extension proposed in the CHI'98 workshop is that it uses the heavyweight extension mechanism of the UML, that is, creates additional concepts at the M2-level (see section II.3.3). This kind of UML extension requires thorough checking for inconsistencies with the remainder of the elements in the metamodel (omitted in Figure III.4 for clarity). The CHI'98 proposal omitted any references to the abstract or concrete syntax, that is, it is not possible to devise mappings between the new semantics introduced and the notation. Furthermore, heavyweight UML extensions require "meta-tool" facilities for tool support and they also impact the interchange formats, for instance XMI (see section II.3.3).

The irrefutable conclusion that use-cases are not a generalized form of task modeling was also corroborated in the ECOOP'99 workshop. Thus, the inclusion of a task model in the CHI'97 and ECOOP'99 workshop frameworks. A task model can convey information about existing tasks and envisioned tasks. Existing task models correspond to the actual way users perform tasks to accomplish their goals. The process of understanding existing tasks is conventionally called task analysis. Envisioned task models correspond to the way users will perform the tasks the foreseen system will support. The process of designing prospective tasks is usually called task synthesis, when the impact in the business processes is relevant it can be referred to work or process-re-engineering (described in the next section).

Both types of task models discussed before convey the information produced in the early stages of the usability engineering lifecycle, as described in sections II.5.1 and II.5.2.2. Therefore, a task model ought to include the artifacts generated for the user-centered activities that encompass understanding and specifying the context of use,

and specifying the user and organization requirements (as presented in section II.5.2.2). From the usability engineering lifecycle (UEL) perspective, this activity includes the tasks of user profiling, task analysis, to some extent usability goal setting, in addition to the work-reengineering task (as described in section II.5.1).

III.3.2.Business (Process) Model

From a traditional Management Information Systems (MIS) perspective a business process is "the unique ways in which organizations coordinate and organize work activities, information, and knowledge to produce valuable products or services" [Laudon and Laudon, 2000]. Therefore, a business model encompasses all the business processes in the context, or related to, the application domain of software intensive systems. This definition is consistent with the one provided by Jacobson and colleagues where "the business process model describes the business processes of a company or organization in terms of its workers, clients and entities they manipulate (...) The business model presents the business from the usage perspective and outlines how it provides value to its users (partners and customers)" [Jacobson et al., 1999]. Therefore, a business model captures the internal operations of a business together with perforce interactions with the outside world – an important issue in human-computer interaction [Nunes et al., 1999].

From a requirements engineering perspective a business model is "a model of the communications and contracts among the participants in the business and the other stakeholders, customers, suppliers and so on" [Graham, 1998]. This model starts with the identification of the different external actors (or agents in the requirements engineering tradition) and their shared goals, also known in the literature as customer value propositions (CVPs) [Jacobson et al., 1995]. However, external goals are insufficient to characterize the goals of a business and, thus, must be complemented with internal goals. Internal goals are those that the organization holds for it's own private reasons and have to do with the underlying cultural and ethical principles [Graham, 1995]. A business model in this perspective can be represented with a mission matrix or grid where the shared and internal goals define one dimension and the stakeholder's roles define another dimension. The business processes are then distributed along the mission grid defining who does what with respect to the different shared and internal goals. Logical roles correspond to abstract job descriptions that unify a coherent set of processes, which can be accomplished using one basic set of skills [Graham, 1994].

Therefore, the requirements engineering approach defines a business as a network of communicating agents. An agent is an autonomous and flexible entity capable of communicating with the external world in a proactive way that can involve exhibiting intelligence and social behavior [Graham, 1995]. Agents in a business communicate through signs and signals, called semiotic acts. In an object-oriented perspective semiotic acts can be represented as messages that can imply data flows and are

directed from the initiator to the recipient. In this perspective agents can also be modeled as special types of objects (actors) that pass messages. According to [Graham, 1995], this approach based on semiotics, is preferable to other approaches because it's not too close to system design, as other alternatives such as Jackson System Development and its OO variants [Graham, 1998]. The author also argues that, besides being one of the most widely used approaches, use-cases are both theoretically and practically flawed for conveying a business model [Graham, 1995].

Some confusion might occur when comparing the concept of a task model with the concept of a business model. However task models are abstractions of user behavior while performing tasks with existing or envisioned systems; whereas a business model represents the business processes from a usage perspective that produces value in terms of services or products for the organization [Nunes et al., 1999]. Not all the activities involved in a business process necessarily involve software intensive systems, in which case both models can eventually match. However, the typical case, which is consistent with the usability engineering perspective (see section II.5.1), involves translating business goals to usability goals that subsequently drive work reengineering.

There is also a relationship between a business model and a domain model. A domain model can be conceived as a simplified variant of a business model where we only focus on the entities that workers manipulate to perform the business processes [Jacobson et al., 1999]. This is also consistent with the HCI perspective were a domain model might include all kinds of user referents that are not germane to the business model [Nunes et al., 1999].

Another important aspect, related to business models, is business process re-engineering (BPR). BPR involves studying the key business processes in an organization with the aim of improving their efficiency, for instance, redesigning process to make their path more proficient. Thus, BPR provides the opportunity to improve existing processes or offer new services to the different stakeholders. From a usability engineering perspective, BPR or work reengineering involves realizing the power and efficiency that automation makes possible; reengineering work to more effectively support business goals; and minimizing retraining by using as much as possible the user's task knowledge, while maximizing efficiency and effectiveness by accommodating cognitive capabilities.

Business (process) models, whether product of business re-engineering or not, convey the information produced in the early stages of the usability engineering lifecycle (see sections II.5.1 and II.5.2.2). Therefore, in a similar way to the task model, a business process models also contains artifacts generated for the user-centered activities that encompass understanding and specifying the context of use, and specifying the user and organization requirements (as presented in section II.5.2.2). From the usability

engineering lifecycle (UEL) perspective, this activity includes the tasks of work reengineering, and to some extent usability goal setting (as described in section II.5.1).

III.3.3.Domain Model

A domain model concerns someone's view of existing or possible reality [Wand et al., 2000]. This definition is based on the ontological perspective, the branch of philosophy dealing with models of reality, which distinguishes between concrete things (entities) and conceptual things (e.g. mathematical concepts). Therefore, the notion of a concrete thing (an entity or an object) applies to anything perceived as a specific object by someone, whether it exists in physical reality or someone's view of reality [Wand et al., 2000]. The ontological perspective is consistent with other definitions in theoretical work on object orientation and human computer interaction. For instance, conceptual models as defined by Martin and Oddel (section II.2.4) and mental models as defined by Norman (section II.1.2.4).

Mental models refer to representations that people construct in their minds of themselves, others and objects in the environment, that is, a collection of entities as defined in the ontological perspective. A domain model simply restricts the entities, which can be considered, to the concepts that apply to a selected area of interest. This restriction, to the context of the system, is consistent with the commonly definition of domain model found in the Unified Process: "a domain model captures the most important types of objects in the context of the system. The domain objects represent things that exist or events that transpire in environment in which the system works." [Jacobson et al., 1999]. However, in the CHI'97 workshops, the authors argued that a domain model "(...) specifies user domain semantics. A domain model is composed of types, which model referents and represent the users' world at a sufficient level of detail to perform user interface design. Users may appear as part of the domain model if they are required to be in the model." [Harmelen et al., 1997].

The main difference, between both perspectives, lies in the restriction found in the CHI'97 vision to the entities required to perform user-interface design. Such restriction is sometimes isolated with a different model called conceptual core model, for instance in the Idiom method [Harmelen, 1996] and in the original version of the CHI'97 framework [Harmelen et al., 1997]. The conceptual core model restricts the domain entities to the subset seen from the end-user perspective, in other words the entities required to perform user-interface design. The conceptual core model can also be conceived as a view, or projection, into the domain model from the end-user's perspective.

Domain modeling concerns the early stages of the usability engineering lifecycle (see sections II.5.1 and II.5.2.2). The domain model conveys information about the user-centered activity of specifying the context of use, and part of the corresponding task of contextual task analysis in the usability engineering lifecycle (UEL). In addition, since

a domain model reflects to some extent the user's mental model, it drives initial conceptual models of the system image. Hence, domain modeling also concerns the activity of designing prototypes and the corresponding UEL task of conceptual model design (see, respectively, sections II.5.1 and II.5.2.2).

III.3.4.Interaction model, User Interface and Interactive System Model

According to the initial CHI'97 definition: "An interaction model provides a specification of the interactive facilities to be used by the users when invoking core functionality in the system." [Harmelen et al., 1997]. Thus, an interaction model provides a description of the semantics of the presentation of the core functionality to the users and the interaction with that functionality (behavior), including interaction styles and techniques.

An interaction style includes the look (appearance) and feel (behavior) of interaction objects and associated interaction techniques from a behavioral (user's) point of view [Hix and Hartson, 1993]. Examples of interaction styles include command entry, menus and navigation, direct manipulation [Shneiderman, 1982], surface interaction [Took, 1990], instrumental interaction [Beaudouin-Lafon, 2000] and so on. Platform specific guidelines describe specific issues related to interaction styles that generally point the way toward consistency and standardization.

There is a close relationship between the interaction styles and the notion of an interaction model. For instance, Beaudouin-Lafon defines an interaction model as "a set of principles, rules and properties that guide the design of an interface. It describes how to combine interaction techniques in a meaningful and consistent way and defines the "look and feel" of the interaction from the user's perspective" [Beaudouin-Lafon, 2000]. Thus, properties of an interaction model can be used to evaluate different design for concrete user-interfaces.

Types in interaction models include abstract and concrete interface objects, such as, selection mechanisms, temporal dependencies, windows, panels, instruments, forms, command, menus, etc. In addition, the objects described by those types can be used as a basis for automatic generation of user interfaces during an implementation phase (as discussed in section III.2.3).

In the original CHI'97 framework there was no specific reference to the separation between the conceptual structure of the interaction and the concrete realization of the user-interface - enforced by the specific user-interface technology and other implementation constraints. This distinction, introduced by [Kovacevic, 1998] and later reinforced in the ECOOP'99 workshop [Nunes et al., 1999], is consistent with the tradition of model-based design of user-interfaces and architectural models of user interface (see sections II.6 and II.7). As a consequence the revised framework, depict in

Figure III.3, introduces a user interface model that corresponds to the concrete and technology dependent objects of its interaction model counterpart.

A user-interface goes beyond the interaction styles and presentation elements used to specify it. The primary role of a user interface is to present to the users the internal functionality provided by the underlying application. According to [Kovacevic, 1998], each user interface is a product the application information requirements and the look and feel requirements. There are many ways the application information requirements can be satisfied, for instance, different interaction techniques can be used to specify an input value, or values can be specified in varying order. A user-interface results when we determine those look and feel requirements, depending on the target user-interface technology or platform. Kovacevic adds that those requirements are optional in a sense that even if they are not fully satisfied (e.g., there is no desired interaction technique available on a target platform and a different technique must be used), the application will still be functional [Kovacevic, 1998]. Hence, the look and feel requirements affect the application usability and not its functionality.

The last model introduced in the CHI'97 and ECOOP'99 workshop frameworks is the interaction system model. In the CHI'97 understanding, an interactive system model aggregates the core model and the interaction model [Harmelen et al., 1997]. Conversely, since there is no distinction between a core conceptual model and a domain model in the ECOOP'99 workshop perspective, the interactive system model refers only to the domain and interaction models [Nunes et al., 1999]. Kovacevic points out that, "notation must facilitate this collaboration by supporting any additional UI related modeling views and providing underlying semantics that ties UI design view construct with OO A&D constructs (in the same manner UML ties existing views)" [Kovacevic, 1998]. In essence, an interactive system model is useful when using an implementation a user needs, *inter alia*, to understand both the semantics of the system and the means of interacting with the system [Harmelen et al., 1997].

The relationship between the three user-interface specific models presented in this section and the usability process frameworks presented in sections II.5.1 and II.5.2.2 is not evident. The interaction models, the user interface models and the interactive system model convey the representations required to support user interface design in later stages of development. Those models either reflect the tradition of model-based design of user-interfaces (section II.7) or the new "movement" towards an integrated view of object modeling and user-interface design discussed in the context of related approaches in the following sections. The usability process frameworks only address the notable exception of prototypes. However, prototypes are illustrations of the system user-interface (whether partial, complete, function or non-functional) and hence are distinct from the software system itself.

III.4. TOWARD INTEGRATED USER-CENTERED OBJECT-ORIENTED METHODS

In the previous sections we reviewed some of the recent developments related to both object-oriented development and user-interface design. The requirements defined in the conceptual frameworks of the CHI'97, CHI'98 and ECOOP'99 workshops [Harmelen et al., 1997; Artim et al., 1998; Nunes et al., 1999] revealed a new understanding for the integration of usability engineering in modern object-oriented software development. This new understanding requires integrated development methods that bridge the gap between software engineering and user-interface design on the common ground of the object-oriented paradigm. This section reviews the existing methods that conform to this new understanding and proposes a new process framework that accommodates those methods.

In a recent book about object-oriented user interface design, Mark van Harmelen [Harmelen, 2001c] refers to development methods that integrated both fields as object-oriented and human-computer interaction methods (OO&HCI). According to the author OO&HCI method can be accommodated in a framework that operates according to four principles, as follows [Harmelen, 2001c]:

- The intervention of HCI design techniques in early phases of the development lifecycle;
- An emphasis on model building and user interface design as the means to establish interactive system scope, functionality and concrete user interface;
- Validation of the developing interactive system design by involving users in evaluation of a range of prototypes throughout the design process;
- Iterative redesign on the basis of prototype use and evaluation to converge on viable, efficient and usable support of users' task-based behavior.

The framework of OO&HCI methods differs from the general user-centered design principles (see section II.5.2.1); notably because of the emphasis on model building for user interface design to establish the system scope, functionality and user interface. This emphasis enables OO&HCI methods to address the fundamental nature of interactive systems, lacking in traditional object-oriented methods. To overcome this deficiency, OO&HCI methods enable specific support for design of use, scope, contents, capability and user interface presentation and behavior [Harmelen, 2001c].

III.4.1.Problems in Traditional Object-Oriented Methods

The motivation for an integrated object-oriented and user interface design approach is that traditional object-oriented methods have severe problems addressing user concerns. Some of those problems were discussed in section III.3 and can be summarized as follows [Harmelen, 2001c]:

- The suitability of conventional use-case modeling as a primary vehicle for conveying user-interface concerns;
- The support for developing a user interface design during object-oriented analysis and design.

Since their introduction in object-oriented software engineering use-cases have become ubiquitous in development practice. Despite the exceeding usefulness of use-cases to structure the application internals and the unifying role in driving development, their support for user-centered activities is deficient. Furthermore, use-cases are recognized to be imprecisely defined leading to multiple interpretations, both in theory and practice, which severely compromises their utility. Evidence of this inadequacy for the purpose of requirements engineering, and user-centered design was discussed in section III.3 and extensively supported in [Constantine and Lockwood, 2001; Graham, 1998; Cockburn, 2000; Harmelen, 2001c]. This system-centric nature of use-cases compromises the focus, throughout development, on the user-centered requirements describing the context of use and defining the added value for the underlying organization. The consequences, compromising the usability of the end products, are manifest in systems that contain functionality that is either hard to use or that provides no results of value to the users.

In traditional OO development, user-interface design is considered a separate activity from both analysis and design. The conventional understanding is that user-interface design is an additional (and separate) OO design level activity, which usually compromises prototyping the presentation of an already defined set of internal functionalities. As van Harmelen points out, "this conventional understanding unfortunately separates too deeply interrelated design activities" [Harmelen, 2001c]. The consequences of this problem are evident at different levels. The architectural baseline developed during OO analysis fails to consider user-interface requirements, which leads too often to sub-optimal user interfaces constructed around a system-centric architecture. Usability problems are repeatedly discovered too late in the development and are either hard (and costly) to repair or unrecoverable at all. Furthermore, failing to represent user-interface architectural concerns, compromises the traceability between the internal and user-interface elements.

III.4.2.Review of User Centered Object-Oriented Methods

As we saw previously, User-centered OO methods, or OO&HCI methods as described in [Harmelen, 2001c], differ from conventional OO approaches incorporating design techniques that help discover, articulate and address users' needs, while supporting user-interface design models and techniques. We only consider user-centered OO methods those that explicitly comply with those requirements, thus, following the understanding of the unifying framework presented in section III.2.2.

The methods reviewed in this section, follow the tradition of OO analysis, design and implementation - as described in the Unified Process [Jacobson et al., 1999]. Furthermore, many of those methods incorporate or maintain compatibility with use-case modeling enabling an important link with the structuring of internal functionality. Beyond these similarities the user centered OO methods described here, vary in many ways. However, they share a list of principles described by van Harmelen and summarized as follows [Harmelen, 2001c]:

- The adoption of a user centered design approach through the integration of HCI design techniques in the early analysis phases and the use of field studies to yield design information about user behavior and the context of use. Also, favoring the UCD tradition of treating users as equal partners in design conveying their knowledge of their own needs, work context and work processes to the design team; and yielding appropriate division of work and functionality between users and interactive systems;
- A very strong emphasis on OO model building and user interface design as the means to establish the scope, functionality and user interface. Specifically the development of an OO conceptual model (depending upon the designers perception of the users' mental models) together with some form of description of envisioned user behavior that shape the interactive system design;
- The possibility of the interaction model informing the design of the domain and conceptual models. Notably through the use of descriptions of user behavior, often task models, that use or are compatible with use-case models;
- Avoidance of prematurely constraining concrete design decisions early in the design process, favoring abstract expressions of user interaction. While adopting an engineering approach that acknowledges and caters for real-world technology constraints while trying to make suitable design trade-offs that preserve the coherence user's conceptual model and the usability of the delivered system;
- Development and validation of the interactive system design by involving users in testing and evaluating a range of prototypes. Iterative redesign to coverage on viable, efficient and usable support of users' task-based behavior;

III.4.2.1.Object, View and Interaction Design (Ovid)

The Object, View and Interaction Design (Ovid) method evolved from early work by a group at IBM involved in the development of the Common User Access (CUA) object-oriented user interface architecture. Work on the CUA architecture (see III.1) began in the late 80s and aimed at exploring the underlying concepts of an object-oriented user interface. From 1989 to 1992 CUA evolved and was widely adopted by IBM and Microsoft products. Ovid was then the result of applying OO analysis and design concepts, together with related tools, to the task of designing user interfaces.

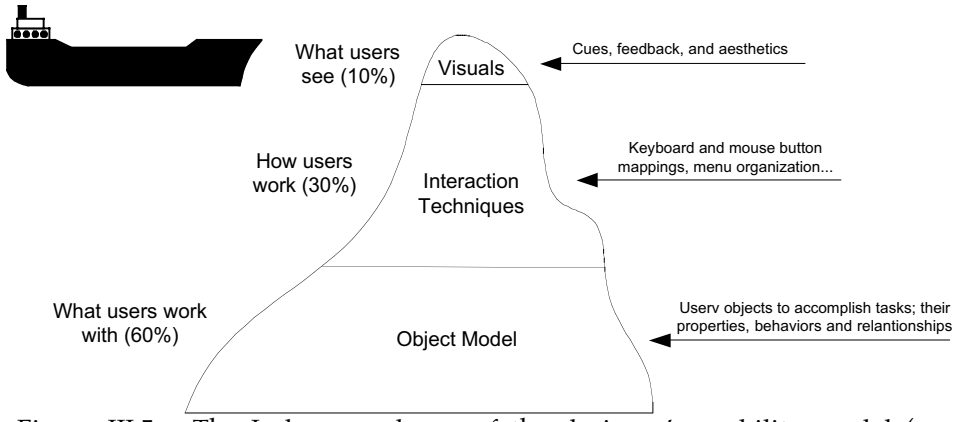


Figure III.5 – The Iceberg analogy of the designer's usability model (source: [Roberts et al., 1998])

Ovid reinforces the idea that visual representations and interaction techniques (the look and feel) are not the most important part of a user-interface. The underlying objects and relationships are the most important aspect of a designer's model, that is, designers should concentrate on the objects, their relationships and behaviors and how they structure the user interface to support the users in their tasks. This idea is proposed in [Roberts et al., 1998] with the Iceberg analogy, depict in Figure III.5.

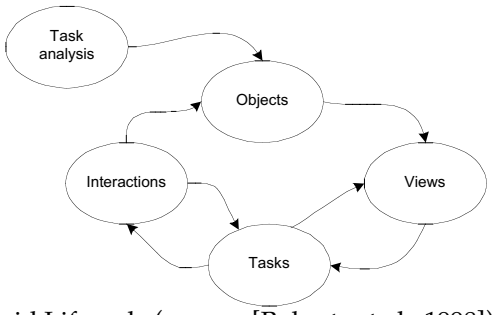


Figure III.6 – The Ovid Lifecycle (source: [Roberts et al., 1998])

The key goal in Ovid is the construction of the designer's model that represents the application from the perspective of the user. This iterative process, depict in Figure III.6, compromises four lifecycle stages that start with task analysis. Ovid is not prescriptive regarding task analysis. The method doesn't provide a specific technique for the purpose of task analysis, instead the author's claim that Ovid can start from any task model built through use-cases, scenarios or any other kind of task modeling

technique. The core iterative cycle of Ovid includes the following steps [Roberts et al., 1998]:

- Identify the objects the user will have to deal with (the object in the user's mental model);
- Suggest some views of those objects that will enable users to interact with the objects needed to perform each task;
- Document the details of how these interactions will occur in the form of new task descriptions;
- Document the details of the specific interactions with individual view and the related objects.

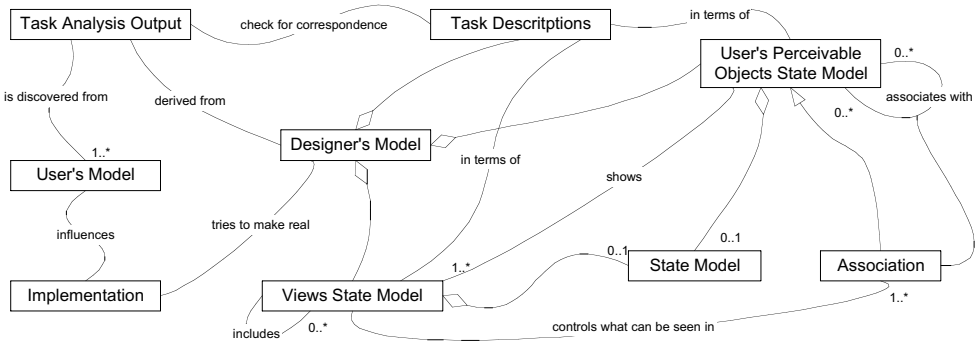


Figure III.7 – Models in Ovid and their Relationships (adaptation of an unpublished meta-model provided by Dave Roberts)

Figure III.7 illustrates the different models in Ovid and their relationships. In the following is a description of Ovid's models (or descriptions) with corresponding notations [Roberts et al., 1998]:

- Task analysis output and task descriptions – as we mentioned previously, Ovid is not prescriptive about model for task analysis, hence the output of task analysis, and corresponding task descriptions, can be any of natural language scenarios, a use-case model, a hierarchical task model, and so on;
- User's Model – the user's model is an object model in the HCI tradition of mental models (see section II.1.2.4). User's models are depict with UML class diagrams;
- Designer's Model - the designer's model is an object model corresponding to the user's model augmented with object views. Views in Ovid present information to users and allow them to use the information to accomplish desired tasks. Views can generally be classified as composed, contents, properties and user assistance or help [Roberts et al., 1998]. The designer's model is an object model in Ovid and is represented with UML class diagrams. Views were originally depict using UML notes but more recently the authors have proposed to use stereotypes of UML classes;
- View state model – view state models in Ovid correspond to the lifecycle of the View objects. This state model is represented in Ovid with both UML statecharts and conventional state tables;

- Implementation – an implementation in Ovid corresponds to the reification of the Designer's model and is represented with UML class diagrams. The reification corresponds to the factoring of the designer's model, including concerns related to the look and feel restrictions of the target platform;

III.4.2.2.Idiom

The Idiom method was firstly proposed in [Harmelen, 1996] as an object-oriented method with an emphasis on specification activities for graphical user interface WIMP design. The first version of Idiom was based on the assumption that object models can convey significant details about the interactive aspects of a GUI-based system. Thus, Idiom enabled user actions to be depicted graphically in interaction sequences.

The author redesigned idiom recently [Harmelen, 2001b] to add upstream design activities, such as scenario generation and subsequent task based design. Idiom doesn't incorporate field studies or explicit participatory techniques, although the author assumes that they are compatible with Idiom.

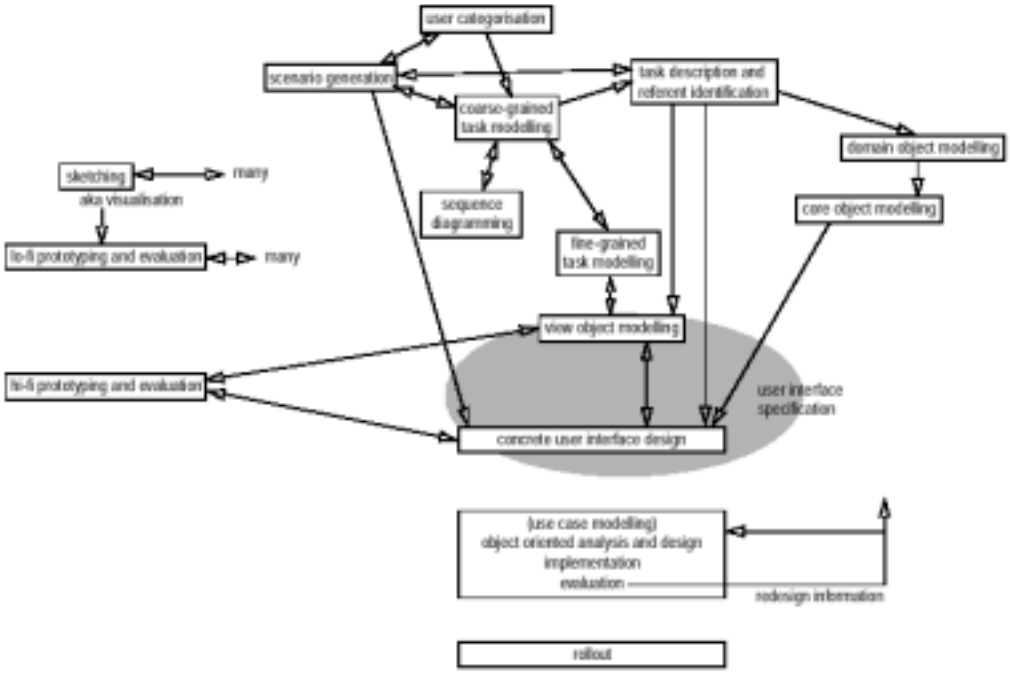


Figure III.8 – The Idiom Process Framework (source: [Harmelen, 2001b])

The overall process of Idiom is illustrated in Figure III.8. Development activities are depict by boxes, and logical or informing dependencies by connections. The development activities in Idiom are as follows [Harmelen, 2001b]:

- User categorization, scenario generation, and task modeling as early user-centered design activities;
- Domain modeling, which corresponds to the object-oriented tradition;
- Core modeling, which corresponds to conceptual modeling in the HCI tradition;

- View modeling, compromising the composition of perceptual entities (views) that provide the context for task execution and how they are dynamically crated and destroyed;
- Designing the concrete details of the interactive system, including the representation of domain objects at the user interface level and how the user interacts with those representations (selection mechanisms, commands, feedback, cursor design, screen layout error handling and help system);

Idiom uses the following models (and descriptions) with corresponding notations [Harmelen, 2001b]:

- Scenarios are concrete textual description in natural language;
- Task models – Idiom is not prescriptive as to the nature of task models. In [Harmelen, 2001b] course-grained task models are represented as structured textual descriptions in natural language. Fine-grained task models are represented with UML sequence diagrams;
- Domain and Conceptual Models – In Idiom both domain and core models are represented with conventional UML class diagrams;
- View Model – although views are referred as object in Idiom, the notation to express the structural aspects of views is not consistent. In early stages views are depict using an ad-hoc notation that renders views as boxes annotated with structured text to indicate the tasks supported. In later stages views appear as extensions of the core model (UML class diagram), including containment relationships amongst them. Semantics for properties of views are not defined. The behavioral of views is expressed with UML statecharts;
- Concrete user interface model – views also appear in the concrete user interface model. However, the additional semantics is provided with interaction sequences that provide a description of the dynamics of the interaction using VDM [Jones, 1986].

III.4.2.3.Usage-Centered Design

Usage centered design is a model based development method based on the initial ideas of essential use-cases, as proposed by [Constantine, 1992]. Usage centered design is described by it's authors as an orderly and systematic method that can be used with any existing software development lifecycle and incorporated into different development technologies, in particular object-oriented development approaches and the UML [Constantine and Lockwood, 1999].

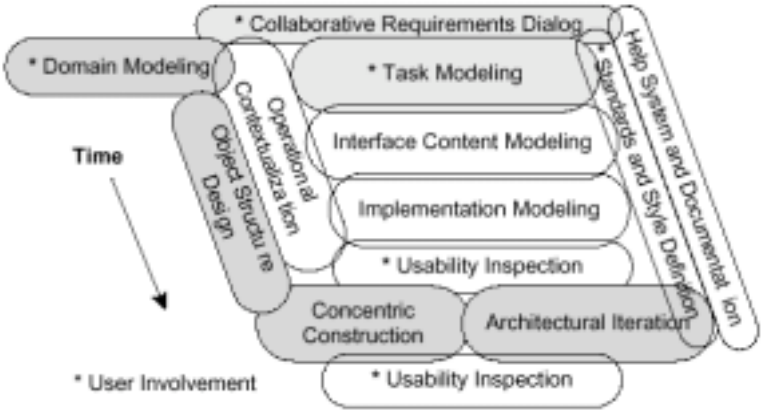


Figure III.9 – Model of the design activities in Usage-centered Design (source: [Constantine and Lockwood, 1999])

Usage-centered design recommends a concurrent engineering process involving different development activities that are carried out in parallel whenever possible (see Figure III.9). The set of activities proposed by Usage-centered Design is as follows [Constantine and Lockwood, 1999]:

- Collaborative requirements dialogue – is a specialized conversation and negotiation between developers and their users (including clients) to establish the requirements of the system to be constructed;
- Task modeling - involves constructing a clear and complete picture of the work to be supported through role and task models;
- Domain modeling – involve developing a representation of all the interrelated concepts and constructs in the application domain;
- Interface content model – involves developing the abstract representations of the various interaction spaces that support the user's tasks;
- Implementation modeling – concerns the concretization of the content model by describing the design of the user-interface and the description of its operation;
- Usability inspection – involves assessing the implementation model for usability problems through testing with end-users;
- Operational contextualization – concerns adapting the design to the actual operational conditions and environment in which the system will be deployed;
- Standards and style definition – involves gathering, documenting and adapting existing standards and guidelines to influence the design. In usage-centered design the standards are also, contrary to the HCI tradition, revised and reviewed by the outcome of other design activities;
- Help system and documentation – concerns developing the help system and documentation in parallel with the other development activities;
- Object-structure design – involves applying object-oriented design techniques in the OO development tradition to structure the underlying object models;
- Concentric construction – is the process of developing working systems in layers as guided by essential use-case models;

- Architectural iteration – corresponds to the maintenance of the sound internal software architecture as successive layers are added to the system by concentric construction;

Figure III.9 illustrates the concurrent design activities in user-centered design. The overlapping between activities reinforces the idea that activities coincide in time and often proceed in parallel. Contrary to the OO development tradition (see section II.4), Usage-centered Design suggests that usage models should drive the whole development process; consequently user-interface activities play a central role of in the process model.

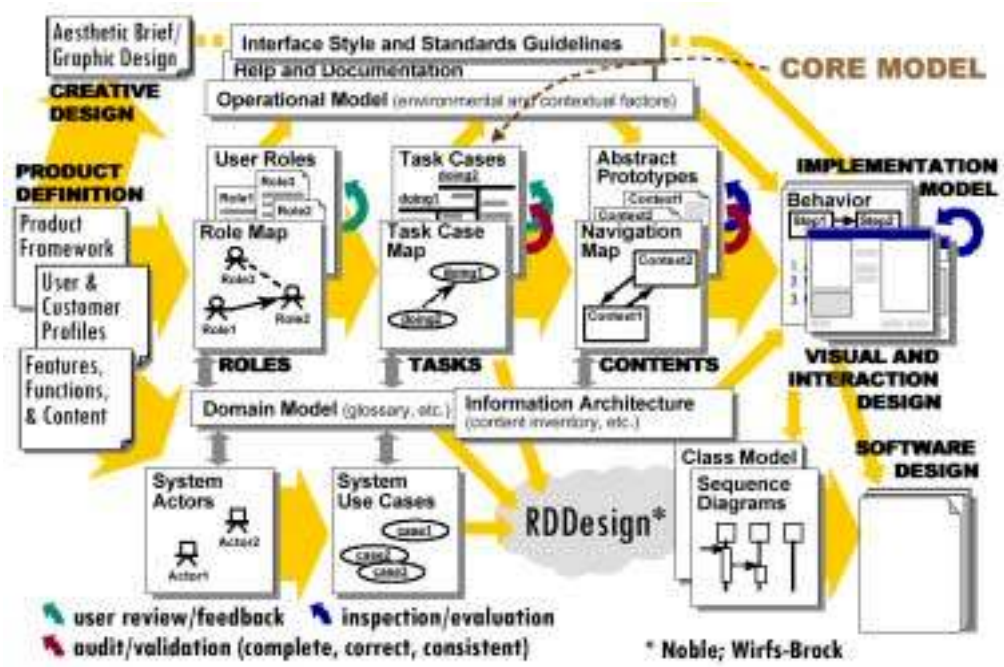


Figure III.10 – Models and logic dependencies in Usage-centered design (source: [Constantine and Lockwood, 2001], updated version provided directly by the authors)

Usage-centered Design is a model driven method that employs a set of abstract models and specific notations. There is no explicit reference to the UML in usage-centered design. The authors adopt a critical attitude towards the UML standard, specifically in what concerns conventional use-case modeling and the usability of the notation (see for instance [Constantine and Lockwood, 2001]). However, the ad-hoc notations suggested in the method are, to some extent, compatible with different UML notations. For the purpose of contrasting Usage-centered design with the other approaches described here, were applicable we mention UML notations and diagrams that can be used to represent the models. Usage-centered design defines the following models [Constantine and Lockwood, 1999]:

- Operational Model – a collection of the salient factors in the operational context affecting the user interface design, including user profiles and environment profiles. User and environment profiles are captured in Usage-centered design using natural language;

- Domain Model –captures the tangible or conceptual entities of the application domain. Domain models in Usage-centered design are represented with object or entity-relationship models, and natural language glossaries for simple cases;
- User role model - captures and organizes the salient or distinguishing characteristics as they affect the use of the system. A user role model is represented using a user role map, which corresponds roughly to a UML use-case diagram with stereotyped relationships between actors (affinity, classification and composition);
- Task Model –represents, in the form of essential use cases, those things that users in user roles are interested in accomplishing with the system. Task models are represented in Usage-centered design with essential use-case narratives (refer to section III.3.1 for the definition) and use-case maps. Use case maps are a collection of essential use-cases and their interrelationships, which can be depicted in the UML as an extended use-case diagram;
- Content Model – is an abstract model of the contents of the user-interface in terms of abstract components (tools and materials) contained within interaction contexts. A content model includes a context navigation map, which is a model of the user-interface architecture showing the interaction contexts and the transitions and connections among them. Navigation maps can be described in the UML as class or object diagrams.

III.4.2.4.Other Methods

The methods described before (Idiom, Ovid and Usage-centered Design) are user-centered OO methods because they comply with the outline provided in section III.4. In particular, all the methods described before provide a user-centered process framework (iterative and involving users) and they all focus on OO model building with specific modeling techniques to support user-interface design.

In this section we briefly review other notable user-interface related methods that by some reason don't fit under the classification of user-centered OO methods described before. Some of the methods presented are simply process framework, while other are specific techniques to assist user interface design in the development process. For a detailed review of the methods in this section refer to [Hudson, 2001; Gulliksen et al., 2001; Harmelen, 2001c].

Graphical User Interface Design and Evaluation (GUIDE)

GUIDE [Redmond-Pyle and Moore, 1995] was originally proposed for the development of large Windows based applications in the late 80s and early 90s in the UK. GUIDE includes a number of user-centered concerns, such as, early focus on users and task analysis, development of a user conceptual model, and design of the user-interface prior to internal system design.

GUIDE proposes the inclusion of a set of user-interface specific activities in the overall development lifecycle. The activities proposed by GUIDE start by the definition of the users and usability requirements, and then follows with a concurrent set of three tasks (model user tasks, model user objects and define style guide), the user-centered core ends with an iterative cycle of design, prototype and evaluate of the graphical user interface (GUI).

Logical User-Centered Interaction Design (LUCID)

Logical User-Centered Interaction Design (LUCID) is a general user-centered process framework developed at Cognetics Corporation by [Kreitzberg, 1996]. The framework follows the user-centered tradition with a stronger focus on prototyping. LUCID identifies six stages: develop the product concept, perform research and need analysis, design concepts and key screen prototype, do iterative design and refinement, implement software, and provide rollout support.

As a management strategy LUCID makes the commitment to user-centered design explicit and highlights the role of usability engineering. A distinctive aspect of LUCID is its focus on key-screen prototypes that incorporate the major navigational paths of the system. This strategy enables a bi-focal design approach, where sections of the user-interface architecture are tested with users through hi-fi prototypes, while keeping the focus on the overall user-interface architecture.

STructured User-Interface Design for Interaction Optimization (STUDIO)

STructured User-Interface Design for Interaction Optimization (STUDIO) emerged at KPMG consultants in the early 90s [Browne, 1993]. Browne's approach is particularly tailored for client-server development, where STUDIO is only applied to the interface intensive clients. STUDIO is not a model-based method but contains many familiar user-centered concepts.

User Interface Modeling (UIM)

User Interface Modeling (UIM) is a method for gathering user requirements that are directly applicable when designing the user interface of an information system [Lif, 1999]. UIM is basically a complement to traditional use-case modeling. UIM specifies an actor model, a goal model and a work model in participatory sessions with end-users, software developers and user interface designers.

UIM doesn't describe a systematic procedure for creating user-interfaces. UIM approach is based on the assumption that user-interface design is a creative process that can't be described in a methodological way. Instead UIM facilitates the design decisions by providing models that defines the user requirements regarding the user-interface.

Entity, Task, Presenter Architectural Framework

The Entity, Task, Presenter (ETP) framework is defines a user-interface architecture with the assumption that the HCI artifacts that form the user-interface descriptions can be classified into three major groups: entities, tasks and presentation elements (presenters) [Artim, 2001]. This classification is an elaboration of the original analysis framework of Jacobson's et al OOSE method [Jacobson, 1992] (see also section II.4.3.2).

The Bridge

The Bridge is a comprehensive and integrated participatory method (see section II.5.3) for quickly designing object-oriented, multiplatform GUIs [Dayton et al., 1998]. A typical Bridge session encompasses three explicit steps.

Part 1 expresses the user requirements as task flows. Here the goal is to translate user needs for the task into requirements that reflect the task flows. The output of this step is a set of index cards and sticky arrows describing what users will do with the new interface.

Part 2 of the Bridge is mapping task flows to task objects. The goal is to map the UI requirements into discrete units of information that users manipulate to do the task (task objects).

Part 3 of the Bridge to map task objects to GUI objects. This third step of the Bridge guarantees that the GUI is usable for executing the task.

III.4.3.A general framework for an Integrated User-Centered Object-Oriented Process

Following a proposal from a position paper at ECOOP'99 workshop, in [Nunes and Cunha, 1999] a suggestion for a UC-OO process framework was considered consensual in expressing the general requirements for integrated methods as described in section III.4. Here we describe this approach, originally proposed in [Nunes and Cunha, 1999] and later reformulated in [Nunes, 1999] and [Nunes et al., 1999].

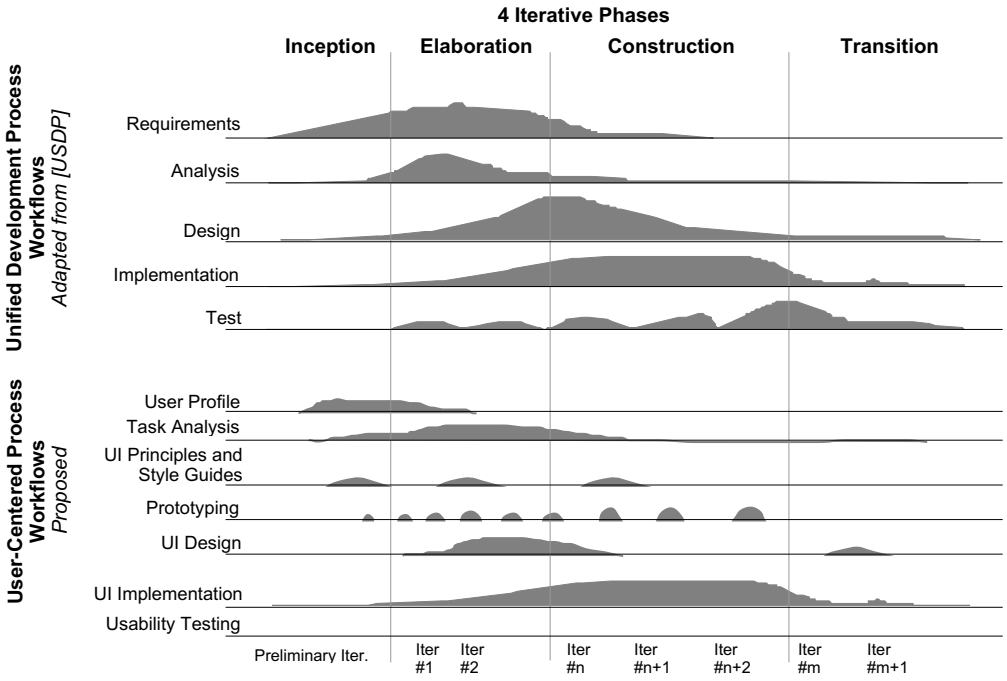


Figure III.11 – The User Centered Object-Oriented Process Framework [Nunes et al., 1999]

The user-centered OO process includes the seven following workflows: User profile, Task analysis, User Interface Principles and Guidelines, Prototyping, User Interface Design, User Interface Implementation and Usability Testing. Figure III.11 tries to depict the effort that is usually required in each workflow as the actual project time goes by.

The workflows of the UC-OO process framework are summarized in the following sections. Relationships with specific model based techniques described in section III.4.2 are included. The aim is to give a broader perspective of a general and *ideal* UC-OO process framework, supported by the existing features of the UML 1.3 standard [OMG, 1999].

III.4.3.1.User Profile

User profile analysis concerns the description of the specific user characteristics relevant for interaction design (e.g., special needs, job expertise, computer literacy and expected frequency of use). It also identifies major user groups required for the task analysis workflow. This workflow will drive almost all design decisions, except those related to technical constraints. The effort in this workflow will typically concentrate on the inception and early elaboration phases.

Major techniques used to profile users are questionnaires and interviews. Traditionally, artifacts produced from this workflow are data summaries, analysis and conclusions. Examples of model-based approaches to user profiling proposed by the

Usage-centered Design method, described in section III.4.2.3. Additional examples are provided in UIM, described in section III.4.2.4.

III.4.3.2.Task Analysis

Task analysis is the process of learning about users, by observing them in action, and building a user-centered model of their work as they actually perform it [Hackos and Redish, 1998]. This process is quite different from asking users questions outside their typical environments. This workflow will be used to set user interface principles and style guides and drive user interface design. The effort in this workflow starts late in the inception phase, builds up during the elaboration phase and can extend to early construction, if the system boundaries are not stable in the first iterations of this phase.

As we discussed in section III.3.1, there is strong consensus about incorporating task modeling in UML. Use cases have been found to be insufficient as general form of task modeling. The heavyweight UML extensions proposed to solve this problem at the CHI'98 workshop failed their purpose, because they required more flexibility than the current UML standard offers for extensibility. The current practice to solve this problem can be characterized as follows:

(i) Not prescribing modeling notations to perform task modeling, therefore enabling the use of traditional HCI techniques (e.g. scenarios, hierarchical task analysis) or conventional use-case modeling. Examples include Ovid and Idiom, described in sections III.4.2.1 and III.4.2.2;

(ii) To extend the conventional understanding of use-case modeling, enabling use-cases to include requirements suitable for user-interface design. A notable example is essential use-cases as initially proposed in [Constantine, 1992] and prescribed in Usage-centered Design (section III.4.2.3) and other approaches (e.g. [Beck and Fowler, 2001]). However, such proposals still require the use of natural language structured scenarios that are not easily related to other modeling constructs. Other proposals in this direction are described in the UIM approach (section III.4.2.4 and [Lif, 1999]) and [Cockburn, 2000];

(iii) To use existing UML diagrams to express task models. This approach is prescribed for task synthesis purposes in Ovid, Idiom and ETP using UML sequence diagrams. Others proposed to use or extend UML statecharts or activity diagrams [Silva and Paton, 2000; Markopoulos and Marijnissen, 2000]. The general problem with using or extending sequence or activity diagrams is related to the capability of such models to depict and enable usable manipulation of moderate size task descriptions (see sections II.7 and V.1.2). Those UML models were provided to express intra and inter-object behavior for the purpose of specifying the system internals. They are recognized to be

extremely complicated to manipulate and use for task analysis when end-user involvement is required on the light of user-centered tradition [Paternò, 2000].

III.4.3.3.User Interface Principles and Guidelines

This workflow concerns gathering, reviewing and adapting relevant user interface design principles, guidelines and standards (see section II.5.1). Principles, guidelines and standards will ensure coherence and consistency across the user interface, driving the user interface design workflow. The effort in this workflow is divided among the three initial phases. At the inception phase general principles are gathered and reviewed from the literature and consulted from experts. In the elaboration phase those principles are adapted to the specific project information produced in the other workflows, particularly user profile and task analysis. Finally in the construction phase, they are evaluated and refined as a consequence of prototype evaluation. The outputs of this workflow are guidelines and standards documents.

Guidelines and standards are by nature textual descriptions in natural language of "how-to-do" solutions; see for example [Apple, 1992; Microsoft, 1995; IBM, 1991].

III.4.3.4.Prototyping

Prototyping is acknowledged to be a major technique in modern software system development. It supports the iterative and incremental nature of the process, enabling the generation, discussion and evaluation of design decisions. Although one can argue that the UP design and implementation workflows can (and should) inherently support prototyping, the UC-OO process isolates this workflow due its conceptual difference in user interface design. The effort in the prototyping workflow builds up through the phases of inception, elaboration and construction. In the inception phase prototyping is typically used to gather information from users and represent high-level ideas of functional organization and conceptual design. At this stage prototyping is based on low-fi techniques (mock-ups, etc.) and used in conjunction with participatory techniques. In elaboration prototyping is used to validate the user-interface conceptual architecture. At this stage both the overall navigational structure and selected subsets of the system are prototyped and evaluated with the end-users. Hi-fi techniques can be used at this stage, although the participatory nature of the evaluation sessions should limit the fidelity of the prototypes, that is, it is well acknowledged that users have problems criticizing prototypes that "look" and "feel" like real systems. At the construction phase prototyping is mainly used to focus on particular problematic subsets of the system, previously unassessed functionality or users.

Model-based descriptions of prototypes are indirectly proposed in the Usage-centered Design method (see section III.4.2.3). Constantine and colleagues have recently

enhanced the idea of abstract prototypes based on abstract canonical components [Constantine, 2000].

III.4.3.5.User Interface Design

An informal survey in user interface programming [Myers and Rosson, 1992] concluded that, on average, 48% of interactive system programming effort was devoted to the user interface. Respondents estimated that the average time devoted to UI development was 45% on the design phase and 50% on implementation time. The user-centered process framework assumes that roughly half of the effort on the original UP design and implementation workflows is transposed to the UI design and implementation workflows when developing an interactive system.

At the user interface design level an UC-OO approach requires support to model the presentation and behavior aspects of the user interface (see section III.3). At the presentation level there is strong consensus about proposing a specific construct to model the contexts where users perform their tasks. Such concept is proposed as an interaction context (Idiom and Usage-centered Design), object-view (Ovid) or presenter (ETP) in all the UC-OO methods described in sections III.4.2.1 to III.4.2.3. This concept is conceived as an enhancement of the interface or boundary architectural element of the OOSE tradition (see section II.4.3.2). Although not semantically equivalent, such approaches point out the need for a modeling concept that captures the "interaction points" as the building block of a model driven interactive system architecture. To model behavioral aspects at the user-interface design level, what is usually known as task synthesis in the HCI tradition, most approaches use the existing UML behavioral diagram. However the same concerns described for task modeling extensions discussed in section III.4.3.2, apply here. A notable exception that aims to follow the model-based tradition, towards automating the generation of the user-interface, is described in [Kovacevic, 1998].

III.4.3.6.User Interface Implementation

At the implementation level the fundamental issues are tool support and traceability between conceptual models and the implementation model. Tool support for task analysis is recognized to be insufficient [Bomsdorf and Szwillus, 1998]. One approach, described in [Lu et al., 1999], is to transform task analysis models into UML interaction models and vice versa. This pragmatic approach enables the integration of specialized task modeling tools to industrial strength tools. Also, the "interaction point" concept seams to enable a clearer management of traceability between conceptual models, i.e., dependencies can be established between (i) the different "interaction points" that compose the navigational structure of the interface, and (ii) the "interaction points" and the other object descriptions belonging to the functional core. Related approaches concern the definition of user interfaces at an abstract level

using a declarative language. Both AUIML [Merrick, 1999], UIML [UIML, 2000] and to some extend Xforms [W3C, 2001] propose an XML (eXtensible Markup Language) [W3C, 2000a] vocabulary designed to allow the intent of an interaction with a user to be defined enabling task designers to concentrate on the semantics of the interactions without having to concern themselves with any particular devices to be supported.

III.4.3.7.Usability Testing

Usability testing aims at identifying problems users will have with the software in all aspects of its use (installing, learning, operating, etc.). Although similar techniques can be used to evaluate prototypes usability testing can only be done when the product reaches the end of the construction phase and in early transition phase. There are different types of techniques that can be used to perform usability testing: formal lab testing, field testing, thinking aloud, walkthrough, inspection, etc. The type of techniques used usually depends on the resources available.

III.5. CONCLUSION

This chapter presented the recent developments in the field of object modeling for user-centered development and user interface design, the specific area of research involved in this thesis.

The main conclusion we can draw from the state-of-the-art in the field is that there is consensus that object orientation can be used to effectively bridge the gap between HCI and SE. As we saw from the brief historical perspective in the beginning of the chapter, object orientation is mainstream in several aspects of user-interface development. Several interaction styles and techniques rely on object-orientation as the underlying implementation technology. Moreover, conceptual and implementation frameworks for interactive systems are grounded on object-orientation. However, it is irrefutable that object modeling for requirements, analysis and design, of the interactive aspects of software intensive systems, is still incipient. In this chapter we reviewed several approaches that try to integrated SE and HCI at this level, but several unanswered problems still remain.

At the requirements level, there is consensus that adequate development of interactive systems relies on understanding the context of use and organizational requirements through user involvement (see section II.5.2). The UC-OO methods presented in this chapter don't provide specific guidance regarding user-involvement and how to systematically translate the information provided by users to the development setting. Participatory techniques (see section II.5.3) are recognized to be an effective solution for this purpose - in particular methods like the Bridge (see section III.4.2.4) have proved to be successful in leveraging user knowledge. However, those participatory methods and techniques, lack specific support on how to translate low-tech materials into the mainstream UML. There is still a gap between the information conveyed in low-tech materials and the more formal artifacts required to specify and model software intensive systems driving implementation. Moreover, the problem is increasingly difficult with iterative development since artifacts are required to sustain user assessment.

Another major issue at the requirements level is the inadequacy of conventional use-case modeling as a general form of task analysis that addresses the concerns of user-centered development. The UML is a language primarily designed to address the problem of developing the system internals and exhibits several problems to address the usability concerns involved in interactive systems. This problem is recognized in all the surveyed UC-OO methods (Ovid, Idiom and Usage-centered Design). However, both Ovid and Idiom are not prescriptive towards task analysis, that is,

they can accommodate both use-case modeling, with the previously mentioned problems; or traditional HCI task analysis, where no notational solution is provided that conforms to the UML enabling semantic interoperability. On the other hand, Usage-centered Design promotes essential use-case modeling as a general form of task analysis, thus solving some of the problems with the system-centric nature of conventional use-cases. Then again, essential use cases, and the related user role maps are not entirely compatible with the UML in their current form.

At the analysis level the major problems bridging HCI and SE is related to the limitations of the conventional OO architectural frameworks, which enable the description of the major structural elements of a software intensive system. There is agreement (although not explicit in the case of Idiom and Ovid) that both behavioral and presentation elements in a UC-OO approach form the elements of the user-interface architecture. Furthermore, Usage-centered design reinforces the idea that user-interface architecture should be central regarding the other development activities. This assumption is grounded in the fact that for the users *the user-interface is the system* and hence all functionality should be layered across an architecture that represents the structure of use and not the internal system functionality. However, and despite the developments in conceptual and implementation architectures for interactive systems (see section II.6), the existing UC-OO methods don't suggest a new architectural framework capable of conveying those issues.

At the design level, the same problems identified with UML support for task modeling, apply to the behavioral aspects of user-interface design (task synthesis). At this level, two of the surveyed methods (Ovid and Idiom) prescribe UML behavioral diagrams (sequence diagrams, statecharts and UML non-compliant state-tables). However, UML behavioral diagrams are recognized to be unusable for complex task sequences and related temporal constraints (see section II.7). Those notations were developed to express inter and intra object behavior and are not adequate for user-interface designer, particularly when their work involves communicating with end-users. On the other hand, Usage-centered design doesn't provide any notation to express the behavioral aspects of the user-interface.

Regarding the presentational aspects of the user-interface, all the UC-OO methods surveyed agree to some extend in a modeling construct that represents an "interaction space" within the interactive system that provides the context for task execution. Although with different semantic meanings, all the approaches promote the utility of modeling the presentation aspects of a user-interface with an interaction context/space/view. However, there are divergences regarding the classification of presentation modeling constructs, including containment and navigational relationships between them. Furthermore, the lack of modeling constructs to adequately specify, document and visualize the presentation and behavioral aspects of user-interfaces, limits the capability to promote tool support and automatic

generation of user-interfaces in the tradition of model-based approaches (see section II.7).

The following chapter described the Wisdom method and the different proposals that solve the problems mentioned previously. Our basic premise is the same argument that drove the adoption of the UML – to enable tool interoperability at the semantic level and provide a common language for specifying, visualizing and documenting software intensive system – applies in leveraging the collaboration between software developers and user interface designers.

IV. THE WISDOM METHOD

"Engineering is not merely knowing and being knowledgeable, like a walking encyclopedia; engineering is not merely analysis; engineering is not merely the possession of the capacity to get elegant solutions to non-existent engineering problems; engineering is practicing the art of the organized forcing of technological change... Engineers operate at the interface between science and society..."
Dean Gordon Brown, Massachusetts Institute of Technology (1962)

This chapter presents the Wisdom method, a UC-OO method developed to address the specific needs of small development teams who are required to design, build and maintain interactive system with the highest process and product quality standards. The chapter starts presenting the application context of Wisdom, small software developing companies (SSDs). We explain the importance of lightweight software engineering methods to support software development in small companies by characterizing those companies, their importance in the software industry, and the major differences from large companies that work with classical lifecycles.

Sections 2, 3 and 4 of this chapter describe the three important components of the Wisdom lightweight software engineering method, as follows:

- The Wisdom process - a software process framework based on a user-centered, evolutionary, and rapid-prototyping model specifically adapted for small teams of developers working in environments where they can take advantage of: (i) enhanced communication both internally and towards external partners; (ii) flexibility in the sense that developers are encouraged to take initiative and are not obliged to follow rigid administrative processes; (iii) control meaning that teams

and projects are easily manageable; (iv) fast reaction in the sense that developers and managers are able to respond quickly to new conditions and unexpected circumstances; (v) improvisation and creativity fostering competitiveness in turbulent development environments where time to market plays a central role.

- The Wisdom model architecture - A set of UML models that support the different development lifecycle phases involved in the Wisdom process. Such models are also required to build interactive system in general (following the general UC-OO process framework presented in the previous chapter). In addition, an original extension of the UML analysis framework (the Wisdom user-interface architecture) that captures the essential elements of well-known and experimented interactive system conceptual and implementation architectures.
- The Wisdom notation - A set of modeling notations based on a subset of the UML and enhanced with UML-compliant extensions specifically adapted to develop interactive systems. The Wisdom notation adapts several recent contributions in the field of OO-UC to the UML style and standard and also proposes new notations to support effective and efficient user-centered development and user-interface design using the UML style as the notational basis. The original notational contributions introduced by Wisdom, enable the description of presentation aspects of user-interfaces, including support for modeling abstract user-interface components, their contents, containment relationships and navigation. In addition, Wisdom supports user-interface behavior modeling with a well-established and original adaptation of one of the most used task modeling notations of the usability-engineering field.

The Wisdom process, architecture and notation are the three major contributions of this thesis. The Wisdom process enables a new understanding of an UML-based process framework and provides an alternative to the Unified Process (UP). The UP is recognized to inadequately support user-centered development and user-interface design. Additionally, the UP framework is not suitable for small software development groups required to rapidly develop high quality products in a turbulent and competitive environment. The Wisdom process overcomes these limitations promoting a rapid, evolutionary prototyping model, which seamlessly adapts the requirements of small software development teams that typically work by random acts of hacking. To accomplish this the Wisdom process relies on a new model architecture that promotes a set of UML-based models to support user-centered development and user-interface design. The Wisdom model architecture introduces new models to support user-role modeling, interaction modeling, dialogue modeling and presentation modeling. In addition, the Wisdom architecture promotes a new analysis-level architectural framework. The new Wisdom UI architecture supports the integration of UI elements has architectural significant structural constituents of the overall system architecture. Finally the Wisdom notation provides a set of new UML extensions that define the new notational constructs necessary to support the models

and architectures required by the Wisdom process and defined in the Wisdom model and UI architectures.

The Wisdom process, architecture and notation are independent contributions that can be broadly applicable in different UC-OO methods. The Wisdom method is a specific proposal for a new UC-OO method that integrates those components. Section 5 of this chapter provides a description of the systematic activities required to develop interactive systems with the Wisdom method. During the description several examples of UML artifacts illustrate the application of the different components of the Wisdom method. The Wisdom method was developed from our experiences working with small software development companies (SSDs), which worked chaotically urging to implementation.

The chapter ends with a comparison between the Wisdom method, including the different models and associated notations, and the other UC-OO methods described in the previous chapter (Ovid, Idiom and Usage-centered Design). The differences are highlighted with respect to the general UC-OO framework presented in the previous chapter, thus outlining the main contributions of Wisdom.

IV.1. APPLICATION CONTEXT: SMALL SOFTWARE DEVELOPING COMPANIES

As the field of software engineering (SE) matures and becomes more comfortable with the methods and processes required to build high quality software in a predicted and controlled manner, there is an increasing concern that the methods developed for conventional large contract-based software intensive systems are not well-adapted to all environments [Laitinen et al., 2000]. Software engineering conventionally targeted large, contract-based development for the defense and industry sectors. Although the SE discipline achieved significant improvements in that area, today modern software development faces completely different problems. In a research survey on the future of Software Engineering (SE), one of the five prominent research objectives is "adapting conventional software engineering methods and techniques to work in evolutionary, rapid, extreme and other non-classical styles of software development" [Finkelstein and Kramer, 2000].

As Laitinen and colleagues point out "The last decade has changed both the character of the software industry and the composition of the companies engaged in development" [Laitinen et al., 2000]. Mass markets, the Internet and entertainment software are today large industry segments subject to intense competition. Small companies are dominating those emerging market segments, and groups of six or fewer developers are now able to create products in excess of 100,000 source lines of code in 18 to 24 months [Laitinen et al., 2000].

In a recent special issue on software engineering in the small, Laitanen and colleagues identified three reasons to advance the study of software engineering in the small [Laitinen et al., 2000]: changes in the software industry, underestimates of the number of small software developing companies, and differences in small group development methods from the standard models. In the following sections we discuss those issues based on previous research on software engineering in the small.

IV.1.1.Underestimating the Importance of Small Software Developing Companies

Small software developing companies are recognized to be the backbone of the software industry. According to the 1992 USA economic census [USCensus, 1997], companies with less than 50 employees accounted for 96,2% of the total companies belonging to the Computer and Data Processing Services sector (SIC 737). They were responsible for 28,3% of the employees in this sector and 26,8% of sales and receipts (23 228,1 millions USD). In Western Europe, where traditionally small companies are

more representative than in the USA, the Software and IT Services market accounts for 106 292 millions of ECU and has the highest average annual growth rate (12.5% and 12.9%) within the Information and Communications Technology (ICT) market [EITO, 1999].

The 1997 Economic Census provided more recent and accurate data for characterizing software development companies. The adoption of the new North American Industry Classification System (NAICS) supports a better way to classify software related activities. The previous SIC classification system included in SIC 737 all the computer related activities (including software development, maintenance, rental and leasing, online information systems, and other computer related services). The new NAICS divides software related activities in a different way. Reproduction of prepackaged software is treated in NAICS as a manufacturing activity; on-line distribution of software products is in the Information sector, and custom design of software to client specifications is included in the Professional, Scientific, and Technical Services sector. These distinctions arise because of the different ways that software is created, reproduced, and distributed. Therefore, for the purpose of better characterizing the impact of small companies in the US software industry we considered companies into two major areas, as follows [USCensus, 1999]:

- 541511 Custom Computer Programming Services - This U.S. industry comprises establishments primarily engaged in writing, modifying, testing, and supporting software to meet the needs of a particular customer. The data published with NAICS code 541511 is comprised in the SIC industry 7371 (Custom Computer Programming Services)
- 51121 Software Publishers - This industry comprises establishments primarily engaged in computer software publishing or publishing and reproduction. Establishments in this industry carry out operations necessary for producing and distributing computer software, such as designing, providing documentation, assisting in installation, and providing support services to software purchasers. These establishments may design, develop, and publish, or publish only. The data published with NAICS code 511210 is comprised in the SIC industry 7372 (Software Publishers).

US Economic Census 1997	Custom Computer Progr. Services	Perc. of Total	Software Publishers	Perc. of Total	Total	Perc. of Total
Total of firms						
Number of Firms	29 248		9 880		39 128	
Establishments	31 624		12 090		43 714	
Paid Employees	318 198		266 380		584 578	
Receipts ($1,000)	38 300		61 699		99 999	
	515		420		935	
Payroll ($1,000)	18 417		18 386		36 803	
	084		784		868	
Single establ. Firms	28 222	96,5%	9 100	92,1%	37 322	95,4%
Multi establ. Firms	1 026	3,5%	780	7,9%	1 806	4,1%
Firms < 50 employees					0	
Number of firms	18 348	62,7%	7 409	75,0%	25 757	65,8%
Paid Employees	107 028	33,6%	63 041	23,7%	170 069	29,1%
Receipts ($1,000)	12 921	33,7%			21 154	
	889		8 233 101	13,3%	990	21,2%
Payroll ($1,000)	6 100 884	33,1%	3 298 093	17,9%	9 398 977	25,5%
Top 50 firms in Sector					0	
Establishments	1 253	4,0%	855	7,1%	2 108	4,8%
Paid Employees	79 775	25,1%	84 999	31,9%	164 774	28,2%
Receipts ($1,000)	10 330	27,0%	35 265		45 596	
	539		678	57,2%	217	45,6%
Payroll ($1,000)	4 720 686	25,6%			12 619	
			7 898 707	43,0%	393	34,3%

Figure IV.1 – Characterization of small software development companies according to the US Economic Census 1997 [USCensus, 1999].

Figure IV.1 summarizes some statistics, drawn from the 1997 US Economic Census, for the two major areas related to software development. The first conclusion we can draw from the figures is that the custom development sector compromises roughly 3 times more firms than the shrink-wrapped sector. However, looking at the number of paid employees the custom sector has a ratio of 10,8 employees per firm, while in the shrink-wrapped that ratio is 26,9. The distribution of receipts by both sectors is also opposite to the number of firms; the shrink-wrapped sector is responsible for 1,6 more receipts than the custom sector. Together both sectors represented almost 100 billion USD in 1997. There is also a considerable effect of concentration in both sectors. The top 50 firms in the custom sector represent roughly 1/4 of the employees, receipts and payroll. In the shrink-wrapped sector the number of paid employees is similar but the total of receipts increases to almost 1/2.

The impact of small companies can be estimated considering companies with less than 50 employees, which is the conventional definition in economic terms both in the US and the European Union [JOCE, 1996]. We can see from the data in Figure IV.1 that small companies account for the large majority of firms in both sectors - 62,7% for the custom sector and 75% for the shrink-wrapped sector. Although that contribution is expected, the number of paid employees is considerable in both sectors with roughly

1/3 in the custom sector and 1/4 in the shrink-wrapped sector. Together there are 25 757 small companies in both sectors (65,8%), which represent 170 069 paid employees (29,1%) and 21 billions of USD (21,2%) in the 1997 economic year.

From this analysis we can conclude that the impact of small companies, from both the custom and shrink-wrapped sectors, is considerable. Furthermore, these figures don't include software groups within user organizations, other sectors highly enabled by software (e.g. online services), temporary workers and the public sector. Moreover, the growth in software employment has increased dramatically in the last few years, and also the impact of the Internet is not evident in the official statistics dating back to 1997.

IV.1.2.Differences in Software Engineering in the Small

Small software development companies (SSDs) face an internal turbulence not common in larger organizations. Environmental turbulence in SSDs is typically caused by the absence of well defined roles that people play in the company, managing multiple projects at the same time, lack of stability in terms of personnel, working in different application domains, and so on. To face these problems, and according to [Dyba, 2000], SSDs require an improvement approach that recognizes:

- The need for a dramatically shorter time frame between planning and actions;
- That planning and actions do not provide all the details of the corresponding implementation;
- That creativity is necessary to make sense of the environment.

Creativity and improvisation play a central role in SSDs because of the environment in which they operate. The environment includes the development context, the characteristics of the market they focus, access to human and other resources, and competition. Improvisation involves dealing with the unforeseen through continual experimentation with new possibilities, to create innovative and improved solutions outside the normal plans and routines [Dyba, 2000]. The current dominant perspective on software development is rooted in the rationalized paradigm that promotes a standardized, controllable and predictable software engineering process. However, software development shares a distinct characteristic with improvisation processes, which is related to the fact that we cannot specify results completely at the outset of the work process [Dyba, 2000]. This means that we can only outline the planned software products and not the processes. Processes based on improvisation differ from the conventional processes because planning and action converge in time – an important issue because time to market often determines competitiveness in small companies [Dyba, 2000].

An improvisation process in SSDs essentially impacts the decisions that have to do with how developers interpret the environment and make decisions in open

situations. For instance, decisions related to modeling, architecture, interaction and resource allocation. A Software Process Improvement (SPI) effort dealing with improvisation involves exploitation (the adoption and use of existing knowledge and experience) and exploration (the search for new knowledge through imitation or innovation). Figure IV.2 illustrates the role of exploitation and exploration in improvement strategies versus organizational size and environment turbulence.

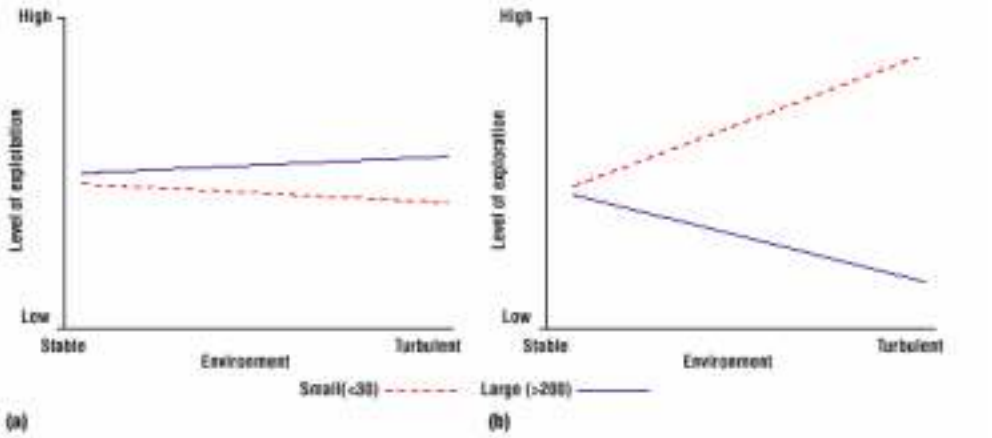


Figure IV.2 – Improvement strategy versus organizational size and environmental turbulence for (a) exploitation and (b) exploration in small and large companies (source [Dyba, 2000])

As we can see from the two graphs in Figure IV.2, large companies react to environmental turbulence increasing their level of exploitation at the cost of reduced exploration. Conversely, small companies react to environmental turbulence increasing their level of exploration at the cost of reduced exploitation. The basic problem that SSDs face is to take advantage of exploitation to ensure short-term results, and to engage in exploration to ensure long-term survival. As Dyba points out "A software organization that specializes in exploitation will eventually become better at using an increasingly obsolete technology, while an organization that specializes in exploration will never realize the advantages of its discoveries" [Dyba, 2000]. Therefore, we need a flexible process model that leverages exploration, thus ensuring that SSDs take advantage of their inherent characteristics and become more competitive. But we also need a controlled process model that enhances the capability to take advantage of exploitation and ensure short-term results. The Wisdom process, described in section IV.2, enables this combination through a controlled evolutionary process model.

Other researchers, involved in applying software process assessment and improvement models like the Capability Maturity Model (CMM) [Paulk, 1995], also concluded that those models are not suitable for SSDs [Brodman and Johnson, 1994; Cattaneo et al., 1995]. The research surveys and case studies available in the literature, describing improvement strategies for SSDs, point out that those companies are concerned with being measured against a model whose requirements they cannot meet. SSDs want to improve, but they also want their own tailored made methods to be accepted by improvement strategies. As we saw previously, SSDs are considerably

different and face completely different environments from large companies for whom SPI strategies were designed. Concerns, like the ones related with the role of improvisation and creativity in SSDs, are ultimately important for SSDs. Furthermore, we must admit that SSDs are at very low maturity levels and that their *ascension* process might, or should in fact, be quite different from what conventional process improvement models recommend (see the Wisdom proposal in Figure IV.5).

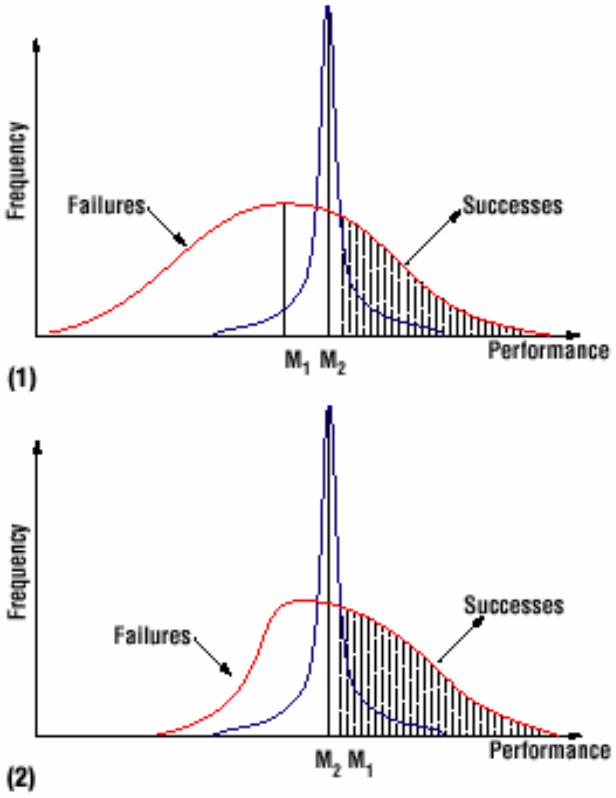


Figure IV.3 – The impact of (1) ability and (2) reliability on performance - M denotes the mean performance of the distributions. (Source: [Dyba, 2000])

Figure IV.3 illustrates why conventional SPI approaches based on "best practice models", like the CMM, are not suitable for SSDs. CMM is based on the argument that standardizing the software process and introducing efficient techniques to developers reduces the time required to accomplish development tasks, thus increasing the productivity, quality and reliability of task performance. Using a model inspired in [March, 1991] Dyba argues that "March's model implies that if the increase in reliability comes as a consequence of reducing the performance distribution's left-hand tail, the likelihood of finishing last among several competitors is reduced without changing the likelihood of finishing first. But, if process improvement reduces the distribution's right-hand tail, it might easily decrease the chance of being best among the competitors despite increases in the organization's average performance" [Dyba, 2000] (see Figure IV.3 - 1). If the goal in SSDs is to increase the competitive advantage, improvement strategies should focus on learning from success to increase the performance of distribution's right-hand tail, while at the same time reducing the left-hand tail by learning from failures (as illustrated in Figure IV.3 - 2).

Therefore, SSDs need an improvement strategy that leverages exploration to ensure competitiveness, but also enables repeatable success. The Wisdom process described in section IV.2, and the underlying improvement strategy depict in Figure IV.5, focus on these two basic premises. On the one hand the controlled evolutionary process model and the standardized notation fosters repeatable success. On the other hand the lightweight nature of the process and the role of participatory techniques enable exploration.

Additionally, in [Nunes and Cunha, 2000c] we discussed the main problems faced by SSDs based on our own experience. SSDs face organizational, cultural, financial, and technical obstacles. In terms of organizational problems, small companies have difficulties forming an internal dedicated process. New practices can affect existing product maintenance and client demand. Management's fear of high costs, delayed time-to-market, and low return on investment manifests in cultural problems. Software engineers often resist new methods, tools, technologies, and standards. Reaching higher maturity levels—required to comply with quality standards—is a long-term commitment. Financial problems often arise because allocating resources for quality groups and hiring consultants are costly. Modern methods and tools are expensive and require training (which temporarily reduces productivity). Finally, small companies can't avoid technical problems: tailoring complex software engineering methods and techniques that are designed for large companies is costly and time consuming. Low-end tools (for example, fourth-generation languages) lack integration and support for management tasks (for example, documentation, configuration management, and so on). Besides raising these problems in small software companies, existing process assessment and improvement models fail to benefit from the strengths of such environments. For instance, small companies are usually more flexible and controllable and they react faster than large companies. Also, communication is usually enhanced in such companies both internally and towards external partners (clients, technology providers, and consultants).

To confirm our experience we conducted an informal survey on SSDs' software engineering practices [Nunes and Cunha, 1998] in Portugal. The study confirmed that SSDs have a strong urge toward implementation. This urge ultimately leads them to chaotic tailored made processes, also known as the Nike® (*just do it*) [Booch, 1996] approach to software engineering. The results of the informal survey showed that Portuguese SSDs are in fact at very low maturity levels, 80% of the surveyed companies didn't use a formal process model, they either *just do it* or use tailored methods. The same study showed that 80% of the companies use 4th generation languages (4GL), either as their sole development tool or in conjunction with 3rd generation languages (3GL). The respondents also confirmed that SSDs are focused on custom or in-house development [Grudin, 1991]. This development context is deeply focused on user interaction, since it usually concerns developing (or adapting) small software systems to meet specific user needs not covered by large shrink-wrapped systems. Therefore there is an increasing importance of human-computer interaction

techniques to support user-interface design, participatory techniques and user-centered design.

SSDs are typically *closer* to end-users and, due to their limited resources, increasingly dependent on the quality of the user interface to satisfy client needs. It is also well known that user-interfaces are responsible for a significant share of the development effort. An informal survey on user interface development [Myers and Rosson, 1992] concluded that, on average, 48% of the code is devoted to the user interface. Respondents estimated that the average time devoted to user interface development during the various phases was 45% on the design phase, 50% on implementation time and 37% on the maintenance phase.

In the following sections we present the Wisdom process, architecture and notation. The Wisdom process defines an evolutionary prototyping development model specifically adapted to provide an improvement strategy for SSDs. The Wisdom process also enables the implementation of the specific activities defined in the Wisdom method in a controlled manner. The Wisdom architecture encompasses a UML model architecture that supports the different models required by the Wisdom method, in agreement with the requirements for user-centered development and user-interface design. In addition, the Wisdom user-interface architecture defines a new architectural framework capable of conveying the architectural significant elements underlying the structure of use. Finally the Wisdom notation is a subset of the UML diagrams specifically adapted to reduce the complexity of the overall language. Moreover, the Wisdom notation extends that reduced subset to include the new requirements for user-centered development and user-interface design.

IV.2. THE WISDOM PROCESS

The Wisdom process defines the basic steps for successfully implementing the Wisdom method in a software development organization, considering the different policies, organizational structures, technologies, procedures and artifacts required to conceive, develop and deploy an interactive software intensive system.

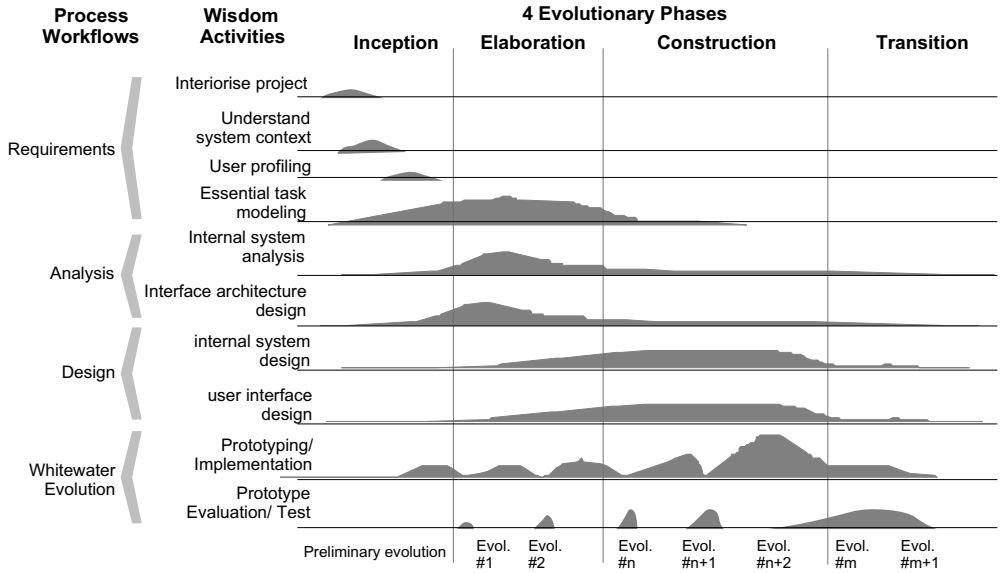


Figure IV.4 – The Wisdom Process Framework

Figure IV.4 illustrates the Wisdom process framework. The process framework is presented in a way similar to the one used to present iterative and incremental processes in the object-oriented software engineering field (see for instance the UP in section II.4.2, Figure II.16). On the top right-hand side of the illustration are the four phases of software development (inception, elaboration, construction and transition), here called evolutionary phases due to the evolutionary characteristic of Wisdom. On the left hand-side are the process workflows. We use the term workflow to describe the activities that software engineers perform and the artifacts they produce in the context of the Wisdom process. At the far left are three of the well-known workflows of software engineering (requirements, analysis and design). We use those terms for clarity when introducing our approach, that is, they are usually recognized by the common software engineer, even those who work by "random acts of hacking". The fourth workflow is renamed in Wisdom *whitewater evolution* and replaces *traditional* implementation. Whitewater is a metaphor, originally used in the Bridge method [Dayton et al., 1998], to illustrate a development process with high resemblance to the intense energy in rapid streams of water where the total (and apparently messy) energy makes the activity as a whole progress very quickly. Evolution complements whitewater because the process evolves in a sequence of incremental prototypes,

ultimately leading to the end product. To the right of the process workflows are the activities performed in the Wisdom method, presented in terms of the devoted effort usually required as time goes by. Finally at the bottom of the illustration we represent time in terms of evolutions. An evolution is defined as a prototype (functional or non-functional) of a software system resulting from intentional (conceptual or concrete), incremental change made with the goal of achieving a desired end product.

The main differences between the Wisdom process and the general UC-OO process framework, described in section III.4.3 (Figure III.11), are related to the higher degree of integration provided by the evolutionary development nature of Wisdom. Contrary to the process framework in section III.4.3, the Wisdom process only makes a clear distinction between internal and user interface development at the levels of analysis and design. During the inception phase all the development activities are combined, meaning that there is a clear acceptance that the development process should be driven by user-centered requirements gathering activities focused on the real world tasks the system will ultimately support. Moreover, the Wisdom process doesn't separate prototyping from actual development. This characteristic is intimately related to the evolutionary nature of the method described in section IV.2.1. Similarly, testing and evaluation activities are concentrated on the iterative deployment of evolutionary prototypes.

Under this broader understanding of a software development process (see section II.4) Wisdom can also be conceived as an improvement strategy for SSDs with chaotic ways of working. Although a chaotic development strategy can take many forms, a tentative representation is illustrated at the top of Figure IV.5. In this representation of chaotic development, the effort is concentrated in the construction and transition phases and corresponds to incremental efforts to increase the functionality of the evolving product. Typically during transition a lot of effort is required to fix several problems (including bugs and usability problems) and shape the functionalities to fit the initial requirements. Therefore the product takes a long time to stabilize and meet the satisfaction criteria of the users, clients and other stakeholders.

SSDs that work chaotically have many problems repeating success, managing and measuring the projects, practicing reuse and product line development. The end products are usually badly structured and manifest high maintenance costs. Although SSDs recognize the drawbacks with chaotic atmospheres, several problems prevent them to improve their processes. As we discussed in section IV.1.2, one of the major problems faced are managerial and practitioner's cultural barriers. Therefore we need a model that seamlessly adapts their environment. The Wisdom process rationalizes *just do it* and builds on what best characterizes small software companies: fast movement, flexibility and good communication. Our approach leverages those characteristics and smoothly evolves the chaotic atmosphere to a controlled evolutionary prototyping model [Nunes and Cunha, 2000c].

In the remaining of this section we briefly present the main characteristics of the Wisdom process philosophy. We define and discuss the important concepts used in the Wisdom method and notation, regarding evolutionary prototyping, essential use-cases and task flows. Where required we contrast concepts and definitions with related approaches both from the object-oriented and usability engineering fields.

IV.2.1.Evolutionary Prototyping

Prototyping has been recognized as an efficient and effective means of developing, exploring and testing software intensive system, and notably their user interfaces. Prototyping is a development approach used to improve software development through the construction of models or simulations (prototypes) of the products or design alternatives to be developed. Prototypes can be used to test concepts, evaluate and explore design alternatives, communicate the look and feel, gather requirements, perform usability evaluation and son on.

Prototyping, viewed from a process perspective, is usually classified as follows [Baumer et al., 1996]:

- Exploratory prototyping – serves to clarify requirements and potential solutions and results in discussion of what should be achieved by a task and how it can be automated by a software system. Resulting prototypes are typically functional and simulate presentational aspects.
- Experimental prototyping – focuses on the technical realization of selected requirements and results in experience about the suitability and feasibility of a particular design or implementation alternative. Resulting prototypes are typically breadboards and functional.
- Evolutionary prototyping – is a continuous process for adapting a software intensive system to rapidly changing requirements and other environment constraints. In evolutionary prototyping all types of prototypes can be developed but pilot systems are of particular importance.

The different types of prototypes that result from the process of prototyping can be classified as follows [Baumer et al., 1996]:

- Presentation prototypes – aim at illustrating how an application automates a given task by focusing on the user-interface aspects of the system.
- Functional prototypes – implement strategically important parts of the user interface and the underlying functionality enabling an effective simulation of a given task.
- Breadboards – serve to investigate technical aspects of a software intensive system, such as architecture or functionality, which involve special risk. Breadboards are usually not intended for evaluation by end-users.

- Pilot systems – are very mature prototypes that can actually be deployed for effective use.

This classification is only one of the possible alternatives for characterizing the products of prototyping. Other popular classifications include vertical/horizontal prototypes and low/high fidelity prototypes [Isensee and Rudd, 1996]. Vertical prototypes correspond to functional prototypes limited in terms of the number of tasks they support. Conversely, horizontal prototypes cover more tasks but with less underlying functionality. Low fidelity prototypes usually refer to limited functionality and limited interaction prototyping efforts that required little cost and effort to develop. Finally high fidelity prototypes are usually fully functional prototypes that highly resemble the end system and that are costly to develop.

Developing prototypes is an integral part of iterative user-centered development [Gulliksen et al., 2001; Preece, 1995; Dix, 1998]. However Wisdom emphasizes the use of prototyping for evolutionary development. There is an important distinction between the conventional understandings of prototyping in iterative user-centered development – usually as an effective technique to communicate and evaluate a given solution - and evolving prototypes to reach the end product. Evolutionary prototyping is an extensive form of prototyping that is usually considered inadequate for conventional software development. The major drawbacks attributed to evolutionary prototyping are related to the impact of non-functional requirements (for instance, those that affect performance) and to the risks of committing to a given architecture or particular solution too early in development. However, evolutionary prototyping is irrefutably considered to be the most effective way of coping with turbulent environments where time to market, changing requirements and competition are crucial factors. Those factors are the ones that typically define the environments in which small software development companies operate (see section IV.1.2).

To take advantage of evolutionary prototyping, without engaging into its well-identified problems, special concerns are required to prevent that end products become badly structured, hence, increasing maintenance costs and preventing reuse. In addition, special care must be taken to prevent the impact of non-functional requirements, like the ones that affect performance. As we discuss in the following sections, Wisdom provides both architectural models that enable control over the structure of the systems (section IV.3) and special techniques, supported by effective notation, that enable exploration of design alternatives and management of non-functional requirements.

IV.2.1.1.From Chaotic Development to Evolutionary prototyping

One can look at the *just do it* approach as only performing the whitewater evolution workflow as a process in itself. In fact, to implement Wisdom in a software

organization, we evolve this chaotic activity introducing the concept of evolution (depict as the second alternative in Figure IV.5). That way Wisdom aims to preserve the whitewater nature of the *just do it* approach, in other words, the importance of maintaining the quick and powerful pace of development in order to seamlessly improve the existing process and prevent unnecessary barriers from managers and the development team. In order to do that the development team is asked to state prototype objectives and perform adequate evaluation of those objectives at the end of the each evolution (the two main activities represented in the second alternative in Figure IV.5). This activity introduces a sense of completion and control, very effective to gain both developers' and managers' support. The key idea is to raise the importance of progress measurement and to support the subsequent Wisdom activities. It is imperative that the development team sees a clear advantage and benefit of those techniques over the random acts of hacking they usually perform.

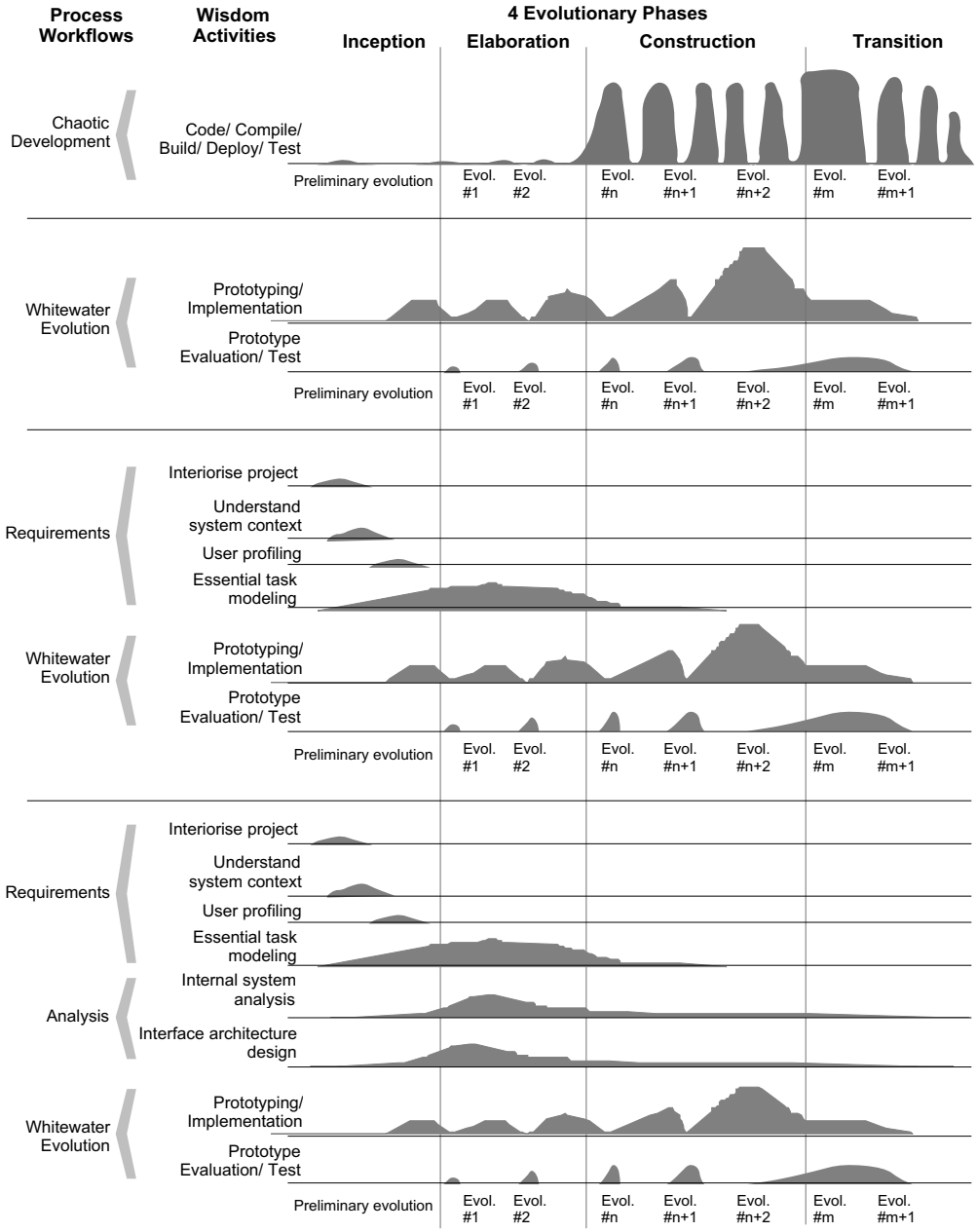


Figure IV.5 – Wisdom as a process improvement strategy. From top to bottom the successive introduction of Wisdom activities leading to the final process.

After successfully introducing the concept of evolution, the development team starts experimenting the Wisdom notation, a subset and extension of the UML (see section 0), through participatory requirements discovery sessions with end-users. At this level the Wisdom process would look like the third alternative in Figure IV.5. Since the major process improvement problems faced by SSDs are on requirements specification and management [Choust et al., 1997], there is a clear advantage starting with this activity. The participatory sessions use a reworked version of the Bridge method (see section III.4.2.4), where a clear mapping between the concepts represented in the low-tech materials and the UML notation is provided. Regarding the process improvement strategy, participatory sessions are a way of seamlessly introducing the notation on a need-to-know basis. Thus we are able to minimize training overload and overcome barriers regarding limited short-term productivity and training overload. Additionally, the model-based nature of Wisdom ensures that creating, managing and maintaining documentation becomes a consequence of the development process itself and not an additional support activity. We call this improvement approach a *friendly trojan horse* approach, because of the *disguised* way the techniques are introduced to the development team.

This goal setting and evaluation cycle, anchored on an UML-based requirements model form the foundation of the Wisdom improvement strategy. Depending on different environment characteristics (team size, experience, maturity, complexity of the project, tools, and so on) we introduce the remaining Wisdom activities, models and associated notation. Always following the same *friendly trojan horse* approach we enable the development team to expand the evolutionary cycle according to their needs. A typical expansion of the Wisdom process corresponds to the fourth alternative depicted in Figure IV.5. This alternative doesn't involve the design activities represented in the full Wisdom process in Figure IV.4. In fact several SSDs that adopted Wisdom, never explicitly perform design activities such as the ones supported by the design-level notational extensions described in section IV.4. This factor is related to the development tools they use, typically 4GLs that lack integration with modeling tools. Without integration between the modeling tools and the actual development tools many companies feel that it's not worth the effort required to create and maintain those design-level models.

IV.2.1.2.Driven by Essential Use-Cases and Task Flows

As we extensively discussed in sections III.3.1 and III.4.1, conventional use-case modeling is not adequate for user-centered development and user-interface design. Although use-cases are an effective tool to structure the internal functionality of the system, driving the development process, they are not a general and flexible form of task analysis. Thus, the current practice in UC-OO methods involves, either extending the conventional understanding of use-cases to include user-centered requirements

(e.g. essential use-cases), or using the existing UML behavior diagrams to express task models (see section III.4.3.2).

Wisdom promotes a combined approach for use-case modeling consisting of essential use-cases to convey the structure of use, and UML activity diagrams do detail that structure in terms of essential task flows. Essential use-cases in Wisdom correspond to top-level tasks and define a way of using a system that is complete, meaningful and well defined to the user [Constantine and Lockwood, 1999]. Therefore, they correspond to goals or external tasks as defined by the usability engineering understanding, that is, a state of the system that a human wishes to achieve (see section II.1.2.2).

Task flows detail essential use-cases through a technology free, implementation independent sequence of desirable and realistic activities required to achieve the goal underlying the essential use-case they describe. Task flows correspond to internal tasks in the usability engineering understanding, in other words, the sequence of activities that a user is required, or believes to be necessary, to do in order to accomplish a goal. The atomic activities in a task flow are internal tasks that involve no problem solving or control structure component (see section II.1.2.2).

The critical difference between the Wisdom use-case modeling approach and the conventional use-case approach [Jacobson et al., 1999], is related to the fact that top level tasks, required by the users to accomplish actual work, drive development and not the inherent internal functionality. Conventional use-cases describe a coherent unit of functionality provided by the system [OMG, 1999], thus a conventional use-case driven process concentrates on units of functionality. An essential use-case driven process is conducted by the real-world tasks that provide real-value to the users. Furthermore, the essential nature of task flows prevents requirement models to contain built-in assumptions about the form of the user-interface of the envisioned system. This approach promotes creativity and leverages novel and simpler solutions for user interface design and the underlying functionality that supports the user-interface.

Wisdom shares the original understanding of essential use-cases as proposed in [Constantine, 1992] and later endorsed in the Usage-centered Design method (see section III.4.2.3). However, Wisdom proposes to use diagrammatic representations of essential task flows depict through UML activity diagrams. This UML-based representation of essential task flows was originally proposed in [Nunes and Cunha, 1999], and fosters participatory gathering of requirements because it is simpler to manipulate than their natural language counterparts defined by essential use-case narratives [Constantine and Lockwood, 1999]. Task flows, emerging from participatory sessions in the form of low-tech materials, are easily translated into UML activity diagrams thus becoming model-based artifacts that software developers can elaborate, transform and relate to other modeling constructs.

Figure IV.6 illustrates the differences between conventional use-case descriptions, as proposed in the Unified Process and the Rational Unified Process (RUP), and structured essential use-cases. The example provided is for a top level task (goal) of cash withdraw in an ATM machine, a common problem addressed in OO modeling textbooks [Rumbaugh et al., 1991; Jacobson, 1992; Kruchten, 1998]. The same example is used as the basis for an extensive discussion of the role of use-cases for user-interface design in [Constantine and Lockwood, 2001].

The first narrative in Figure IV.6 is directly translated from the RUP textbook and describes the example ATM use-case as a numbered sequence of steps. The system-centric nature of this example is obvious. Discussing the same example Constantine and Lockwood point out "The narrative exemplifies a systems-centric view: With one exception, each step begins with the system side of things" [Constantine and Lockwood, 2001]. In fact, this conventional use-case narrative is full of assumptions about design level decisions for the system that is yet to be developed. For instance, the reference of cards and PINs for user authentication. Moreover, the narrative is full of rhetorical details that are not essential for the purpose of illustrating the interaction, and which difficult the usefulness of the representation.

The second example in Figure IV.6 represents a responsibility driven use-case narrative that avoids the confusion between the user actions and the system responsibilities [Wirfs-Brock, 1993], therefore making the user-interface boundary evident [Constantine and Lockwood, 2001]. However, this example still includes assumptions about the implementation of the user-interface, for instance the reference to technology details. This kind of use-case narrative is superior to the conventional description promoted in conventional OO modeling but still inadequate as an effective informing description for user-interface design.

The third example in Figure IV.6 represents an essential use-case narrative (see section III.3.1) for the same ATM use-case. In the words of Constantine and Lockwood, "this example is dramatically shorter and simpler that the concrete use-case for the same interaction because it includes only those steps that are essential and of intrinsic interest to the user" [Constantine and Lockwood, 1999]. The essential use case narrative is problem-oriented and leaves open many possibilities for the design and implementation of the user interface. For instance, user identification could be implemented with retinal scan or voice recognition. Moreover, the potential of this narrative to accommodate different interaction styles and user-interface technologies is far superior from their concrete counterparts.

Conventional Use-case: Withdraw Money

(Transcription from [Kruchten 1999, pp. 96-97])

1. The use case begins when the Client inserts an ATM card. The system reads and validates the information on the card.
2. System prompts for PIN. The Client enters PIN. The system validates the PIN.
3. System asks which operation the client wishes to perform. Client selects "Cash withdrawal."
4. System requests amounts. Client enters amount.
5. System requests account type. Client selects account type (checking, savings, credit).
6. System communicates with the ATM network to validate account ID, PIN, and availability of the amount requested.
7. System asks the client whether he or she wants a receipt. This step is performed only if there is paper left to print the receipt.
8. System asks the client to withdraw the card. Client withdraws card. (This is a security measure to ensure that Clients do not leave their cards in the machine.)
9. System dispenses the requested amount of cash.
10. System prints receipt.
11. The use case ends.

Structured Use-case: Withdraw Money

(Originally proposed by [Wirsf-Brock 1993])

User intentions	System responsibilities
Insert card in ATM	
	Read Card
	Request PIN
Enter PIN	
	Verify PIN
	Display option menu
Select option	
	Display account menu
Select account	
	Prompt for Amount
Enter amount	
	Display amount
Confirm account	
	Return Card
Take card	
	Dispense cash if available

Essential Use-case: Withdraw Money

(Transcription from [Constantine and Lockwood 1999, p. 105])

User intentions	System responsibilities
Identify self	
	Verify identity
	Offer choices
Choose	
	Dispense cash
Take cash	

Figure IV.6 – Three alternative use-case descriptions for the Withdraw cash use-case: (i) Conventional, (ii) Structured, (iii) Essential

Figure IV.7 illustrates the Wisdom diagrammatic approach to essential task flows for the same example of the ATM cash withdrawal use-case. In Wisdom we take advantage of the evident superiority of essential use-case narratives but adapt the concept to a diagrammatical representation to foster user participation and enable a better integration with the UML. Therefore, we combine essential use-case narratives with participatory driven task flows as proposed in the Bridge method [Dayton et al., 1998]. At the far left is the outcome of a typical participatory session where the withdraw cash use-case is worked-out with end-users through manipulation of index cards (activities) and sticky arrows (dependencies between activities). To the right is the corresponding UML activity diagram that results from the translation of the sticky arrows and index cards (see section III.4.2.4) produced in the participatory session.

The differences from the original essential use-case narrative depicted in Figure IV.6 are as follows:

- Wisdom **essential task flows** support multiple success and failure scenarios through the use of decisions and guards, depicted as diamonds both in the Bridge and UML activity diagrams. Multiple scenarios lead to the well-known scenario explosion problem, which can be solved in use-case modeling with subordinate use-cases, extensions and variations [Cockburn, 1997]. There are many techniques to support extensions in structured prose. For instance, control conditions (if, continue, etc.) are used in essential use-case narratives, numbered scenario fragments are used in conventional use-cases, and other approaches even use one description per scenario. Although the alternatives are equivalent from a broad, formal point of view, their expressive power varies [Cockburn, 1997]. On the one hand, control structures introduce composed constructs that require some formal knowledge. On the other hand, scenario fragments can lead to large narratives that are sometimes hard to read due to extensive cross-references.
- Representing activities and dependencies with index cards and sticky arrows fosters communication with end-users. Low-tech materials, like the ones used in Wisdom participatory sessions, provide "an equal opportunity work surface" where users, independently of their computer experience, can express themselves equally [Dayton et al., 1998]. Moreover, pictorial representations are easier to manipulate than structured narratives, users can combine and recombine the activities and their dependencies more freely than textual representations.
- There is a clear mapping between the output of the participatory sessions, based on the Bridge task flows, and UML activity diagrams. As illustrated in Figure IV.7, index cards correspond to UML activity constructs and sticky arrows correspond to UML dependencies between activities. As a result, the development team can feed the result of participatory sessions into more formal UML model elements that are manageable by modeling tools and easily related to other modeling constructs. The translation process provides a good opportunity to add extra information regarding non-functional requirements (depict as UML notes), including usability considerations, design trade-offs or other constraints.
- The development team can recurrently use the task flows in subsequent participatory sessions and apply different techniques to identify task object and relate them to GUI objects (Parts 2 and 3 of the Bridge). Moreover, essential task flows can be used to perform usability testing at each Part of the Bridge method, as described in [Dayton et al., 1998]. Since there is seamless mapping between the output of participatory techniques and the UML activity diagrams, the development team can take advantage of tool support to manage the changes introduced by usability testing.

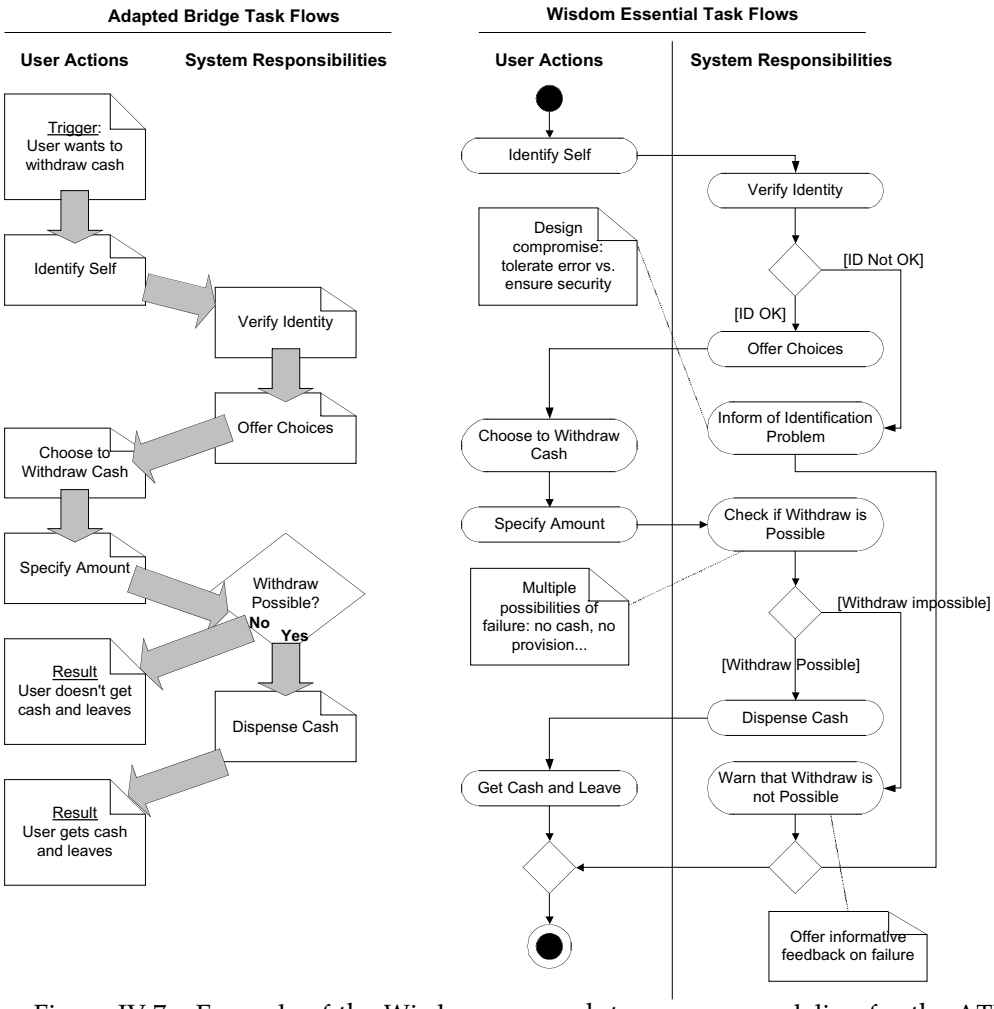


Figure IV.7 – Example of the Wisdom approach to use-case modeling for the ATM withdraw cash example: (i) left-hand side – an adapted Bridge task flow produced in a participatory sessions and (ii) right-hand side – an essential task flows expressed as an UML activity diagram.

The role of Wisdom essential task flows is not restricted to capturing and expressing requirements in participatory sessions. Essential use-cases and task flows in Wisdom drive the evolutionary prototyping process (see section IV.2.1), by intentionally guiding development in terms of the real-world tasks that add real value to the users. Development progresses in terms of evolutions that support an increasing number of essential use-cases, task flows and the corresponding underlying functionality. Thus, the well-known problem of providing functionalities that don't add value (a problem usually known as "featurism") is minimized. Moreover, essential use-cases and task flows serve as containers for non-functional requirements (as illustrated by notes in Figure IV.7) and provide the major input for finding and specifying, interaction and analysis classes.

IV.3. THE WISDOM ARCHITECTURE

One of the more important goals in software development is to establish the architectural foundation of the envisioned system. As we discussed in section II.4.3, a software architecture involves the description of the elements that built the software intensive system, the interactions among those components and the patterns that guide their composition [Shaw and Garlan, 1996]. Despite following the same principles in the above definition, a characterization of interactive system architecture should include other issues related to the interactive nature of the system. However, definitions in the literature are diverse in breadth and purpose.

Artim describes user interface architecture as "an approach to development processes, organization and artifacts that enables both good UI practices and better coordination with the rest of development activities" [Artim, 2001]. This highly conceptual description focuses mainly the process of building an interactive system and the need for collaboration and coordination between HCI and SE practice. Paternò gives a more traditional (in the software engineering sense) definition: "the architectural model of an interactive application is a description of what the basic components of its implementation are, and how they are connected in order to support the required functionality and interactions with the user" [Paternò, 2000]. However, this definition still lacks some key aspects of software architectures, like reuse and pattern composition. Kovacevic adds reuse concerns and some key aspects of model-based approaches observing that such architecture should "maximize leverage of UI domain knowledge and reuse (…) providing design assistance (evaluation, exploration) and run time services (e.g., UI management and context-sensitive help)" [Kovacevic, 1998]. Finally Constantine and Lockwood point out “the term user interface architecture refers to the overall structure of the user interface (…) how all these things are integrated into a complete system that makes sense to the user” [Constantine and Lockwood, 1999].

We consider that an architecture for interactive systems involves the description of the elements from which those systems are built, the overall structure and organizations of the user interface, patterns that guide their composition and how they are connected in order to support the interactions with the users in their tasks.

This characterization of interactive system architecture can be recursively decomposed into parts that interact through interfaces and are described with different models concerning different criteria, for instance, granularity, nature of the entities and usage. The two major architectural descriptions for interactive systems are discussed in section II.6. Conceptual models, for example PAC [Coutaz, 1987] and

MVC [Golberg and Robson, 1983], concern conceptual entities (e.g., the notion of dialogue control) and how the entities relate to each other in the design space. Implementation models, for example Seeheim [Pfaff and Haguen, 1985] and Arch [Bass, 1992], concern software components (e.g., a dialogue controller) and the relationship between the components.

Both conceptual and implementation models of interactive systems have adopted the separation of concerns between the entities that model the task domain and those involved in the perceivable part of the system (also promoted in the meta-model described in section III.2.2). On the one hand, this distinction enables the entities that depend on the task domain (functional core) to be reused with different interpretations and rendering functions. On the other hand, interpretation and rendering entities define a software component (the user interface) that can be maintained independent from the functional core [Coutaz, 1993]. Due to practical engineering issues this separation of concerns is further refined in implementation-oriented models.

The following sections discuss the two original architectural descriptions promoted in the Wisdom method. Both architectural descriptions build on the best practice models described in sections II.4.3 and II.6, thus providing an effective bridge between the understanding of software architecture descriptions in software engineering, and the requirements for the purpose of developing interactive systems.

IV.3.1.Wisdom Model Architecture

As we discussed in section II.2.1, a model is an abstraction of a physical system with a certain purpose. UML models have two goals: to capture the semantic information of a system through a collection of logical constructs (e.g. classes, associations, use-cases, etc.) and to present such information in a visual notation that can be manipulated by developers. Models can take different forms, aim at different purposes and appear at different levels of abstraction. However, they should be geared towards a determined purpose. For instance, as we discussed in section II.2.1, models can guide the thought process, abstract specification of the essential structure of a system, completely specify the final system, partially describe the system or exemplify typical solutions [Rumbaugh et al., 1999]. From this perspective, model architecture defines a set of models with a predefined purpose. Various examples of model architectures are described in sections II.4.3. However, as we discussed in section III.2.2, they don't comply with the necessities required by interactive systems.

The Wisdom model architecture specifies the different models of interest to effectively develop an interactive system under the framework of the Wisdom method. This model architecture guides the though process in Wisdom from the high level models built to capture requirements to the implementation models that fully specify the final system. According to the requirements for interactive system model architecture

discussed in section III.3, the Wisdom model architecture leverages both the internal and user interface architectures. Hence, we solve the problem in traditional software engineering approaches were the focus goes only to the organization of the internal components and their relationships - the internal architecture of the software system. Following the best traditions of the user-centered approach, the Wisdom architecture enables the focus on the external user interface architecture since early project inception. Like the internal architecture is more than a set of internal components, the user-interface architecture is also more than a set of user interface components. In Wisdom the external architecture not only concerns individual user interface components but also (and essentially) the overall structure and organization of the user interface.

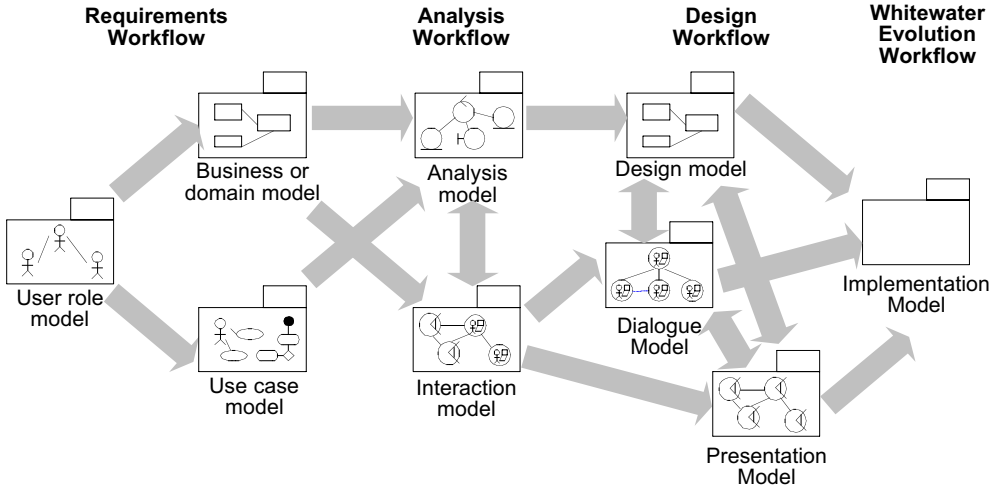


Figure IV.8 – The Wisdom Model Architecture

Figure IV.8 illustrates the Wisdom model architecture. Eight models are developed in Wisdom - excluding the implementation model, which represents all the artifacts required to deploy the system (databases, code, user interface, etc.). Models are roughly aligned with respect to the major conventional workflows defined in OO software engineering: requirements, analysis and design. Whitewater evolution replaces conventional implementation to highlight the importance of the evolutionary prototyping nature of Wisdom (see section IV.2.1).

At the far left in Figure IV.8 is the user role model. This model follows the original perspective of Constantine and Lockwood's user role map as described in section III.4.2.3 [Constantine and Lockwood, 1999]. The user role model captures and organizes the salient or distinguishing characteristics of the users as they affect the use of the system. User roles represent types of users, including the collection of their interests, behaviors, responsibilities and expectations in relation to the system. The Wisdom user role model is represented using UML use-case diagrams with stereotyped relationships between actors (affinity, classification and composition), corresponding to an adaptation of their original semantic definition in Usage-centered Design (in section IV.4.1 we describe the corresponding UML extensions). The user role model is developed during early inception according to the user-centered best

practice of early focus on user needs (see section II.5.2). Information in the user role model can be captured using different usability techniques (site visits, focus groups, surveys, and so on), but in Wisdom we promote the use of focus groups in participatory sessions. The user role model impacts all the following models, specifically the domain and/or business model and the use-case model.

At the top left hand side of the figure are the domain and/or business model. The domain model captures the most important types of objects in the context of the system (see section III.3.3). When the business context, in which the system is to be deployed, is significantly complex; or when business process reengineering is a primarily concern, the domain model can be complemented with a business model. The business (process) model describes how the organization coordinates the work activities, information and knowledge to produce the valuable products or services (see section III.3.2). In Wisdom the domain model is captured using class diagrams. The business process model extends the domain model with additional information captured through activity diagrams. In addition, a business use-case diagram can represent the structure of the organization in terms of business actors and business use-cases. Business actors represent the role that someone or something in the environment can play in relation to the business organization. They are related to business use-cases that define the types of sequences of work steps that produce a result of perceivable and measurable value to an individual actor of the system. Business actors are distinct from user roles, defined in the user role model. While business actors are roles from the perspective of the business organization, user roles represent users from a usability perspective. Business modeling entities used in Wisdom are the ones defined in the UML profile for business modeling (see section II.3.3). The information in the domain and business process models is captured through participatory sessions in early system inception. The domain or business process models impact primarily the interaction and analysis models.

The remaining inception model is the use-case model. A use case model specifies the services a system provides to its users [OMG, 1999]. As described in section IV.2.1.2, Wisdom enriches that specification detailing use cases with essential activity diagrams, hence, focusing on user needs for the tasks reflected in task flows. The use case model is represented in Wisdom with use case and activity diagrams. To avoid scenario explosion (see section IV.2.1.2) we use UML decisions in activity diagrams to ensure there is only one essential task flows per essential use case. The information in the use-case model is highly influenced by the user role model and the domain or business models. Use-cases and essential task flows are built in participatory sessions with focus groups. There is no obvious logical dependency between the business, domain and use case model, but as the arrows (logical dependencies) in Figure IV.8 suggest, they all influence the following models.

The requirement models (business, domain and use case) are an external view of the system described in the language of the customer. The analysis model is an internal

view of the system described in the language of the developers. This model structures the system with stereotypical classes (and packages if required) outlining how to realize the functionality within the system. Therefore, the analysis model shapes the internal architecture of the system defining how different analysis classes participate in the realization of the different use cases. To specify the analysis model in Wisdom we use the UML analysis framework described in section II.4.3.2. This framework in expanded with an original proposal for a new interaction framework described in section IV.3.2. That way Wisdom enables the relationship between the internal architecture and the user-interface (external) architecture. The information in the analysis model corresponds to a technical activity carried out by the software development team. The analysis model is primarily built during system elaboration in the analysis workflow. The analysis model influences directly the design model in the design workflow.

As consistently mentioned throughout the previous chapter, there is common agreement over the clear advantage to separate the user interface from the functional core in conceptual and implementation architectural models for interactive system. In section III.2.3 we discussed that such conceptual (not necessarily physical) separation leverages on user-interface knowledge in providing design assistance (evaluation, exploration) and run-time services (user interface management, help systems, etc.). This approach should support implementation of internal functionality independent of the user interface fostering reuse, maintainability and multi-user interface development, an increasingly important requirement with the advent of multiple information appliances [Norman, 1998]. The Wisdom interaction model is an external view of the system from the user interface perspective. This model structures the user interface identifying the different elements that compose the dialogue and presentation structure of the system (look and feel), and how they relate to the domain specific information in the functional core. The different elements that compose the Wisdom user-interface architecture are described in detail in section IV.3.2. To specify the interaction model in Wisdom we use the corresponding UML notational extensions described in section IV.4.3. The information in the interaction model corresponds to a technical activity carried out by the user-interface development team. The interaction model is mainly built during the analysis workflow of the elaboration phase. The interaction model influences the presentation and dialogue models devised in the construction workflows.

The Wisdom models in the design workflow reflect the same separation of concerns between the internal functionality and the user interface at a lower level of abstraction. The design model defines the physical realization of the use cases focusing on how functional and non-functional requirements, and other constraints related to the implementation environment, impact the system. Hence, the design model is more formal than the conceptual analysis model and specific to a given implementation. To specify the design model we can use any number of stereotypical language dependent classes.

The dialogue model specifies the dialogue structure of the interactive application, focusing on the tasks the system supports and the temporal relationships between tasks. The dialogue model specifies task level sequencing and provides relationships that ensure multiple interaction space consistency, while mapping between domain specific and user-interface specific formalisms. There is general agreement that dialogue models are fundamental models in user interface design because they enable developers to specify the dialogue structure of the application avoiding low-level implementation details (see section II.7). The Wisdom dialogue model serves this purpose refining the user-interface architecture and ensuring the separation of concerns between the user interface and the functional core, and also between the presentation and dialogue aspects of the system. To specify the Wisdom dialogue model we use an UML extension based on the *ConcurTaskTrees* [Paternò, 2000] formalism. This extension and the main features of the formalism are discussed in section IV.4.4.

The presentation model defines the physical realization of the perceivable part of the interactive system (the presentation), focusing on how the different presentation entities are structured to realize the physical interaction with the user. The presentation model provides a set of implementation independent entities (interaction spaces) for use by the dialogue model, hence, leveraging independence of the interaction techniques provided by the user interface technology (e.g. UI toolkit). Interaction spaces are responsible for receiving and presenting information to the users supporting their task. Interaction spaces are typically organized in hierarchies and containment relationships can occur between them. According to [Nunes et al., 1999] an interactive system' architecture should "support the separation of the user interface specifics (look and feel) from its (conceptual) specification, independent of the type of interaction and technology". Such separation leverages automatic user interface implementation from conceptual (abstract) models [Nunes et al., 1999]. We specify the presentation model using a set of UML extensions described in section IV.4.5.

IV.3.2.Wisdom User-Interface Architecture

The Wisdom user-interface architecture is an elaboration of the boundary/entity/control analysis pattern described in section II.4.3.2. This original architectural pattern, initially proposed in [Nunes and Cunha, 2000a], addresses the refinement and structuring of the requirements defined in the early user-centered models. The Wisdom UI architecture encompasses three types of stereotyped classes: entity, task and interaction space.

As we discussed in [Nunes and Cunha, 2000a] the original proposal for new UI architecture introduces a user-centered perspective at the analysis level. Despite all the requirements for interactive system architecture (discussed in sections II.6 and III.2.2), the new architecture model should comply with the following criteria:

- Build on user interface design knowledge capturing the essence of existing successful and tested architectural patterns;
- Seamlessly integrate the existing UP analysis pattern, while leveraging cooperation, artifact change and ensuring traceability between usability engineering and software engineering models;
- Foster separation of concerns between internal functionality and the user interface, not only in its intrinsic structure, but also enabling the co-evolution of the internal and interface architectures;
- Conform to the UML standard both at the semantic and notation levels, that is, the artifacts required to express the architecture should be consistent with the UML and its built-in mechanisms;

The information space for the Wisdom UI architecture is depicted in Figure IV.9. The new information space differs from the information space of the OOSE [Jacobson, 1992], Unified Process [Jacobson et al., 1999] and UML standard profile for software development processes [OMG, 1999], introducing two new dimensions for the dialogue and presentation components. In addition, the presentation dimension of the UP is restricted to non-human interface. Note that the information dimension is shared between the two information spaces, leading to a total of five dimensions if we consider both information spaces as a whole. Such combined information space spans the analysis model and the interaction model, described in the previous section.

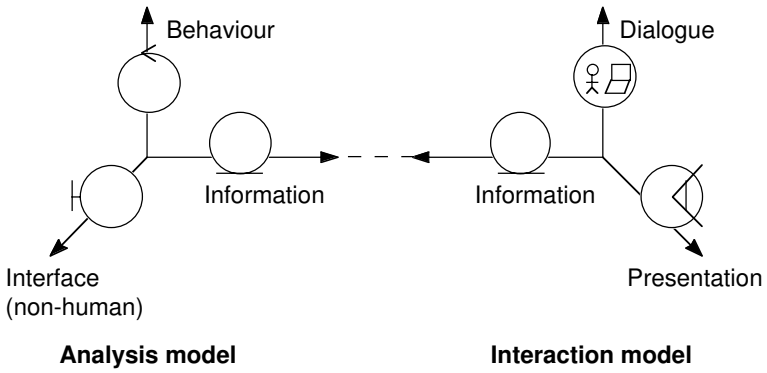


Figure IV.9 - The Wisdom user-interface architecture

This Wisdom UI architecture encompasses the information, dialogue and presentation dimensions of the information space, clearly mapping the conceptual architectural models for interactive systems described in section II.6. Accordingly, the internal architecture accommodates the existing analysis dimensions, also including the shared information dimension. Note that the presentation dimension in the analysis model is reduced to capture the interface (not the presentation) to external systems (system actors). This way we are able to tie the internal and user-interface architectures, leveraging the required separation of concerns, while maintaining the necessary relationship amongst them. Moreover, the two information spaces accommodate the domain knowledge of both the OO and usability engineering communities.

The Wisdom UI architecture, like the MVC, PAC and UP analysis architectures, is a conceptual architecture pattern. Therefore it should not take into consideration design or implementation criteria, like the Seeheim and Arch models do (see section II.6). However, the Wisdom architecture is at a higher granularity level of the MVC and PAC models, hence, it will eventually suffer subsequent reification at design and implementation. Therefore the Wisdom architecture should support such reification, maintaining qualities like robustness, reuse and location of change; while leveraging the mediating nature of the domain adapter and interaction toolkit components of the Arch model (see Figure II.24). This process is typically achieved through precise allocation of information objects (entity classes) to domain adapter and domain specific components at design and implementation time (see II.7). Such allocation enables semantic enhancement (dividing or joining objects in the domain adapter component) and semantic delegation (enhancing performance by preventing long chains of data transfer to objects in the domain specific component) [Coutaz, 1993]. The same applies to the interaction toolkit component, at this level with presentation objects (view classes).

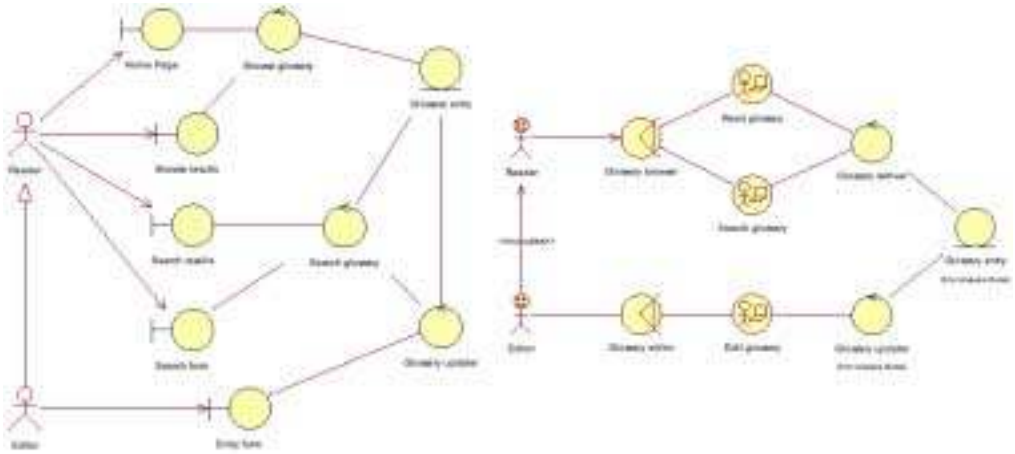


Figure IV.10 – Application example of the conventional OO analysis framework versus the Wisdom UI architecture: left-hand side - transcription of a solution provided in [Conallen, 1999]; right-hand side - the new solution based on the Wisdom UI architecture.

Figure IV.10 illustrates the differences between the Wisdom UI architecture and the conventional architectural descriptions used in the OO field. To the left-hand side of the figure is the analysis conceptual architecture provided in [Conallen, 1999] for a glossary web application. The architecture in the example supports three use-cases (read glossary, search glossary and edit glossary entry). As we can see from left-hand model in the figure, the conventional solution doesn't separate the user-interface from the internal functionality. For instance, *browse glossary* and *search glossary* are two control classes that contain both the business logic required to browse and search glossary entries and the structure of use required to perform those tasks. Furthermore, the conventional model contains built-in assumption about the user-interface technology and the interaction styles used to implement the user-interface of the glossary application. For instance, the boundary classes *Home Page*, *Search form* and

Entry form are obviously indication a form-based interaction style and a web-based user-interface. Assuming technology constraints at the analysis level suggests that this architecture could not support a different technology or interaction style - for example a Java applet – therefore compromising the potential for reuse.

The right-hand side of Figure IV.10 depicts the solution for the same problem based on the Wisdom UI architecture. The Wisdom architecture clearly supports the well-known best practice of separation of concerns between the internal functionality and the user-interface specifics. The advantages are evident, not only the structure of use related to reading, searching and editing the glossary is contained in specific classes (the *read glossary*, *search glossary* and *edit glossary* task classes), but also the structure of the internal functionality becomes simpler because it doesn't have to contain the user-interface behavior. Moreover, there is a clear separation between the presentation of the user-interface (*glossary browser* and *glossary editor* interaction space classes) and the structure of use. The resulting architecture is therefore, simpler (both in terms of the user-interface and the internal functionality), more robust (changes in the presentation of the user-interface don't impact the structure of use and the internal functionality, and vice-versa), and more reusable (the structure of use can be reused with respect to different implementation technologies). Finally the Wisdom UI architecture seamlessly maps different implementation architecture. For instance, assuming typical three-tier implementation architecture for a web-application, interaction spaces and task classes are candidates for client-side components, whereas control classes are candidates for the middleware tier and entities for the data tier.

The Wisdom model architecture (see Figure IV.8) defines a set of UML models that enable user-centered development (user-role model and use-case model) and user-interface design (presentation model and dialogue model). The Wisdom UI architecture (see Figure IV.9) extends the conventional UML analysis framework enabling the description of architectural significant elements that describe the structure of use. In the following section we describe the UML notational extensions required to support those models.

IV.4. THE WISDOM NOTATION

According to a recent study based on charts of concepts, UML version 1.1[2] has 84 basic concepts and 149 diagram concepts, leading to an overwhelming total of 233 concepts [Castellani, 1999] (see Figure IV.11). As we discussed in section II.2.3, UML predecessors (notably OMT, OOSE and Booch) were methods, but the UML is a process independent language. Thus the language includes an extensive number of concepts to enable tailoring for different requirements and development contexts.

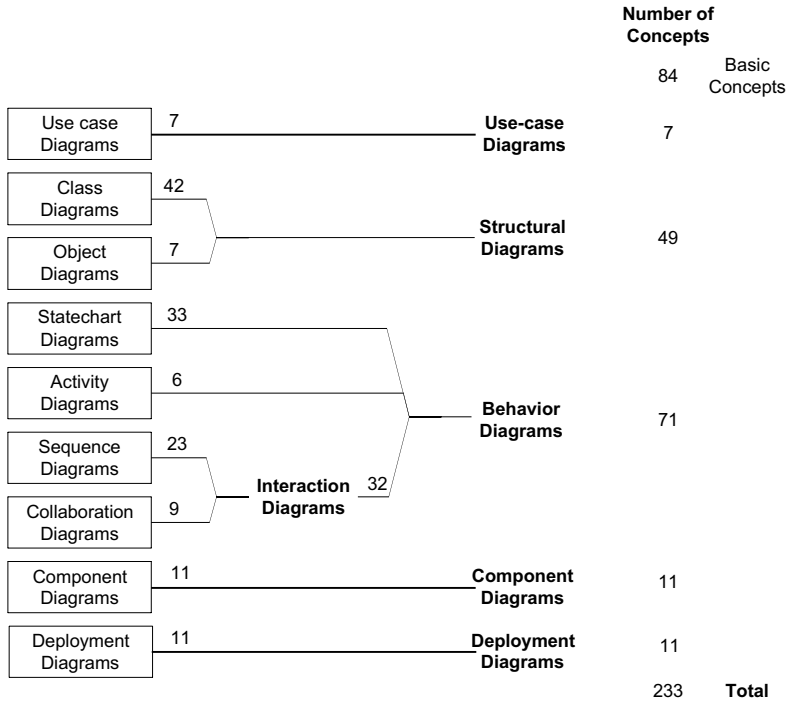


Figure IV.11 – UML Diagrams and Associated Concepts (estimation based on [Castellani, 1999])

The Wisdom notation involves both a subset of the different UML modeling constructs and a set of extensions in the form of a UML profile – the Wisdom profile. A profile is a UML lightweight extension mechanisms that involves a predefined set of stereotypes, tagged values, constraints and notation icons that collectively specialize and tailor the UML for a specific domain or process [OMG, 1999]. The Wisdom profile is therefore a collection of stereotypes, tagged values and notation icons that specialize the UML to support user-centered development and user-centered design.

[2] Castellani's study with charts of concepts refers to UML version 1.1. The author didn't update the study to the current UML version (1.3) because there are no significant changes in the overall number of concepts.

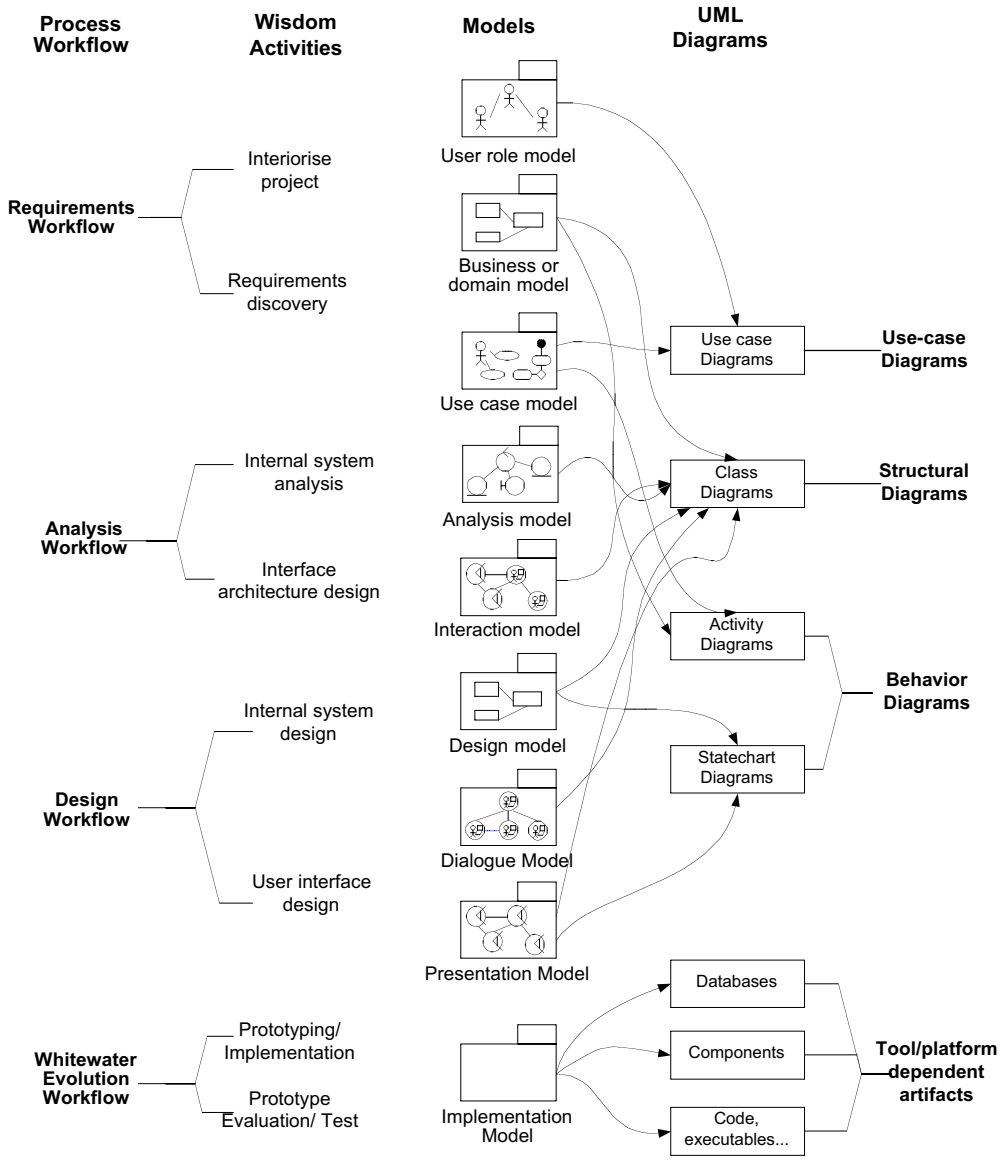


Figure IV.12 – Process Workflows, Activities, Models and Diagrams in Wisdom

Figure IV.12 represents the four process development workflows described in section IV.2, and the corresponding Wisdom activities, models, and diagrams used to specify the different models. As depicted in Figure IV.12, Wisdom in its full form is based on eight models and uses four types of diagrams (use-case and class diagrams and two types of behavioral diagrams). According to the mentioned study about the UML concepts [Castellani, 1999], we estimate that this selection of diagrams and concepts is approximately 39 basic concepts and 29 diagram concepts, leading to an overall total of 68 concepts (29% of UML 1.1 total number of concepts). The Wisdom profile extends the UML with a number of extra concepts (the extensions proposed in this section), but they are limited in number and closely related (stereotyped) from existing UML concepts. Moreover, Wisdom doesn't fully explore all the concepts in each diagram in Figure IV.12; some of the concepts are for advanced modeling purposes and are seldom used. Therefore, the extra concepts introduced by Wisdom compensate the number of advanced and seldom used concepts in the UML diagrams.

In the following sections we define a set of UML extensions, originally proposed in [Nunes and Cunha, 2000b], that support user-centered development and user-interface design. We present the extensions according to the different models defined in the Wisdom model architecture described in section IV.3.1: requirements, analysis, interaction, presentation, dialogue and design. The aim is to describe the notation in relation to the development lifecycle stages were the extensions are more likely to be applied. The presented extensions, together with the standard modeling constructs related with each model in Figure IV.12, collectively define the available constructs required to convey the information in each model.

IV.4.1.UML Extensions for the User Role Model

As we discussed in section IV.3.1, Wisdom uses the original perspective of Usage-centered Design user role maps to convey the information for user profiling. The user role model is a representation of the various user roles and their salient or distinguishing characteristics as they affect the use of the system [Constantine and Lockwood, 1999]. The relevant and salient characteristics in user roles are as follows [Constantine and Lockwood, 1999]:

- Incumbents – aspects of the actual users playing a given role, including important categories such as domain and system knowledge, training, experience, education, and so on;
- Proficiency – how usage proficiency is distributed over time and among users in a given role (e.g. novice, intermediate, expert);
- Interaction – patterns of usage with a given role (e.g. frequency, regularity, predictability, concentration and intensity of interaction);
- Information – nature of the information manipulated by users in a role or exchanged between users and the system (e.g. origin of information, direction of flow, volume and complexity);
- Usability criteria – relative importance of specific usability objectives with respect to a given role (efficiency, accuracy, reliability, learnability, rememberability, user satisfaction and so on – see section II.1.3);
- Functional support – specific functions, features, or facilities needed to support users in a given role;
- Operational risks – type and level of risk associated with the user's interaction with a system in a given role (e.g. life critical systems, safety requirements, etc.);
- Device constraints – limitations or constraints of the physical equipment facilitating the interaction between the user and the system in a given role (e.g. screen size, type of input devices, etc.);
- Environment – relevant factors of the physical environment in which a user in a role interacts with the system (e.g. lightning and noise conditions, mobile use, etc.).

The Wisdom role model is conveyed with UML use-case diagrams, which represent the relationships among user roles in terms of affinity, classification and composition, in a way consistent with the understanding proposed in Usage-centered Design. A user role is one kind of relationship between users and the system, defining a collection of user interests, behaviors, responsibilities, and expectations in relation to the system [Constantine and Lockwood, 1999]. The concept of user roles, as the original authors recognize [Constantine and Lockwood, 1999], is substantially similar to the concept of actor originally proposed in the OOSE method [Jacobson, 1992] and later endorsed in the UML standard [OMG, 1999]. However, there is one important difference, as Constantine and Lockwood point out "actors, in the original definition, include other nonhuman systems – including hardware and software – interacting with the system (…) colloquially "actor" refers to the person playing a part, and "role" refers to the part being played (…)" [Constantine and Lockwood, 1999].

Although the difference between user roles and UML actors is semantically subtle, there is a clear advantage to distinguish between human actors and system actors. This distinction is already provided at the analysis level with the Wisdom user interface architecture (see section IV.3.2) and, therefore, there is an obvious advantage to separate the different types of interaction (human and nonhuman) as early as possible in the development lifecycle. Furthermore, the salient and distinguishing characteristics of both types of actors are evident when we look at the different usability criteria described before.

The following UML extensions define the Wisdom adaptation to convey the information in the user role model. Concepts are adapted from the original proposal of Constantine and Lockwood [Constantine and Lockwood, 1999], but reflect the Wisdom perspective, notably in what concerns the pragmatics of adapting a had-hoc notation into the UML. The Wisdom UML extensions for the user role model are as follows:

- <<essential use-case>> is a UML class stereotype defining the specification of a sequence of user intentions and system responsibilities, including variants, that a human actor can perform, interacting with the system. Essential use-cases are technology free and do not contain any unnecessary restrictions or limiting assumptions regarding specific implementation details reduced to its minimal form of expression;
- <<human actor>> is a UML class stereotype defining a coherent set of user roles that users play when interacting with a system defined in terms of essential use-cases. User roles define a set of distinguishing and salient characteristics, including level of domain knowledge, experience, proficiency, usability aspects and other criteria. A human actor is always an abstraction of an end-user that actually interacts with the application, and excludes other stakeholders not involved in explicit interaction (e.g. clients, customers, domain experts, etc.). A user role is one kind of relationship between users and the system, defining a collection of user

interests, behaviors, responsibilities, and expectations in relation to the essential use-cases.

- <<system actor>> is a UML class stereotype defining a coherent set of system roles that an external system plays when interacting with use-cases. A system actor has one role for each use case with which it communicates.
- <<resembles>> is a UML association stereotype between human actors or essential use-cases that defines a similarity of an unspecified nature.
- <<specializes>> is a UML association stereotype between essential use-cases or human actors denoting that some essential use-cases or human actors are subclasses (subcases or subactors) of another, inheriting characteristics of the superclass (supercase or subactor).
- <<includes>> is a UML association stereotype between essential use-cases or human actors denoting that some essential use-cases or human actors are composed of others. One essential use-case can "use" or depend on other essential use-cases including the sequences of activities that define the other essential use-cases. One human actor may "include" or be composed of other human actors.

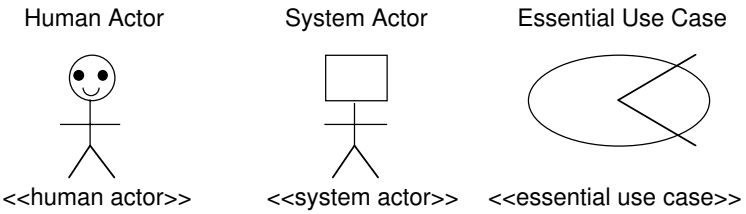


Figure IV.13 - Alternative notations for the class stereotypes of the Wisdom user role model.

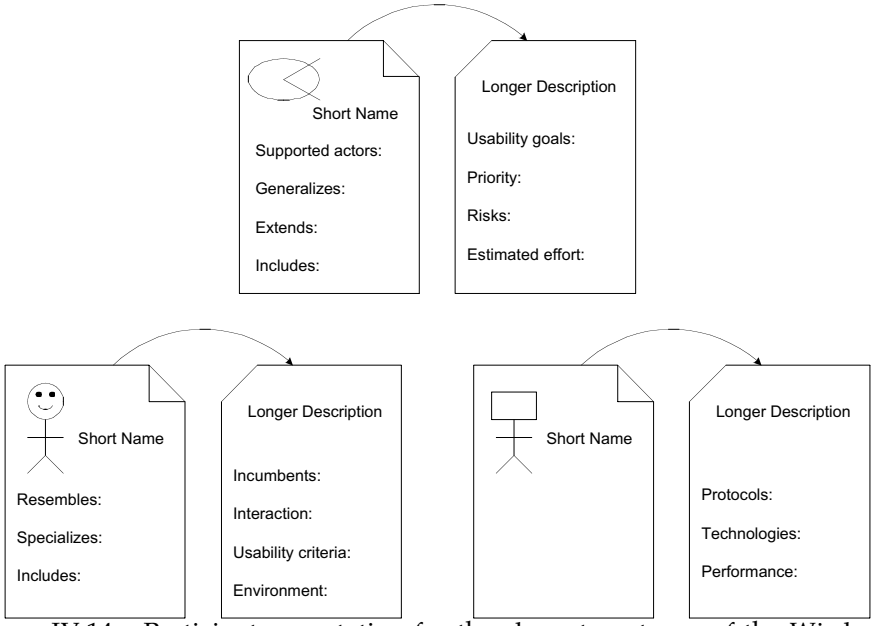


Figure IV.14 – Participatory notation for the class stereotypes of the Wisdom user role model

Figure IV.13 illustrates the notational icons defined by the Wisdom profile for the user role model.

As we discuss in sections IV.2.1.1 and 0, Wisdom user-role models emerge from participatory sessions. The structure of the index cards used for each class stereotype is depicted in Figure IV.14. Associations between modeling elements (human actors, system actors and essential use cases) are conveyed in the front of the index cards. In the back of the index cards additional information about each modeling element is captured. That additional information includes a minimal set of the relevant and salient characteristics for human actors (see the detailed description in the beginning of this section); protocols, technologies involved and performance information in system actors (additional information common in system-to-system interaction); and finally usability goals, priority, risks and estimated effort for essential use-cases.

For essential use-cases the additional information concerns usability goals (see section II.5.1) that define the minimal acceptable user performance and satisfaction criteria; priority in terms of relative importance and value-added for end-users and business objectives; estimated risks associated with the implementation of the use-case (including impact of non-functional requirements); and finally an estimation of the effort required to implement the essential use-case. The additional information captured for essential use-cases enables the development team to directly combine usability criteria and process management information. Since essential use-cases drive the development process, the development team can prioritize development in terms of the added value to the end-users (see section IV.2.1.2). The additional information depicted in the participatory notation for the user-role map can be translated into the UML through stereotyped attributes.

IV.4.2.Revised UML Extensions for the Analysis Model

As we mentioned in section IV.3.2, the Wisdom user-interface architecture expands the understanding of the stereotypes defined in the UML standard profile for software development processes. At the notational level the Wisdom notation introduces some subtle but important changes in the conceptual definitions of the class stereotypes.

The modified definitions introduced in the Wisdom profile (where required partial definitions are retrieved from [OMG, 1999] and [Jacobson et al., 1999]) are as follows:

- <<Boundary>> class stereotype – the boundary class is used, in the Wisdom user-interface architecture, to model the interaction between the system and external systems (non-human). The interaction involves receiving (not presenting) information to and from external systems. Boundary classes clarify and isolate requirements in the system's boundaries, thus isolating change in the communication interface (excluding human-interface). Boundary classes often represent external systems, for example, communication interfaces, sensors, actuators, printer interfaces, APIs, etc.
- <<Control>> class stereotype – the control class represents coordination, sequencing, transactions and control of other objects. Control classes often

encapsulate complex derivations and calculations (such as business logic) that cannot be related to specific entity classes. Thereby, control classes isolate changes to control, sequencing, transactions and business logic that involves several other objects.

- <<Entity>> class stereotype – the entity class is used to model perdurable information (often persistent). Entity classes structure domain (or business) classes and associate behavior, often, representing a logical data structure. As a result, entity classes reflect the information in a way that benefits developers when designing and implementing the system (including support for persistence). Entity objects isolate changes to the information they represent.

IV.4.3.UML Extensions for the Interaction Model

The elements of the interaction model are interaction classes, defined as stereotypes of UML class constructs. The three stereotypes proposed in the Wisdom user-interface architecture are as follows:

- <<Task>> class stereotype – Task classes are used to model the structure of the dialogue between the user and the system in terms of meaningful and complete sets of actions required to achieve a goal. Task classes are responsible for task level sequencing, consistency of multiple presentation elements and mapping back and forth between entities and presentation classes (interaction spaces). Task classes encapsulate the complex temporal dependencies and other restrictions among different activities required to use the system and that cannot be related to specific entity classes. Thereby, task classes isolate changes in the dialogue structure of the user interface.
- <<Interaction space>> class stereotype – the interaction space class is used to model interaction between the system and the human users. An interaction space class represents the space within the user interface of a system where the user interacts with all the functions, containers, and information needed for carrying out some particular task or set of interrelated tasks. Interaction space classes are responsible for the physical interaction with the user, including a set of interaction techniques that define the image of the system (output) and the handling of events produced by the user (input). Interaction space classes isolate change in the user interface of the system, interaction spaces are technology independent although they often represent abstraction of windows, forms, panes, etc.

The UML profile for software development processes also defines association stereotypes. Although, the Wisdom profile doesn't change the semantics of those association stereotypes, they can be applied to the new class stereotypes introduced before, as follows:

- <<Communicate>> is an association between actors and use cases denoting that the actor sends messages to the use case or the use case sends messages to the actor. It can also be used between boundary, control and entity. In addition it can

be used between actor and boundary, with the new specific constraint that actor is an external system. Communicate can also be used between entity, task and interaction space. In addition it can be used between actor and interaction space, with the specific constraint that actor is human. The direction of communication can be one way or two ways;

- <<Subscribe>> is an association between two class states that objects of the source class (subscriber) will be notified when a particular event occurs in objects of the target class (publisher). Subscribe can be used from boundary, entity and control to entity. In addition, subscribe can also be used between task and entity. The direction of subscribe is one way.

The alternative notations for the class stereotypes defined in the Analysis and Interaction models are depict in Figure IV.15 (see also Figure IV.9).

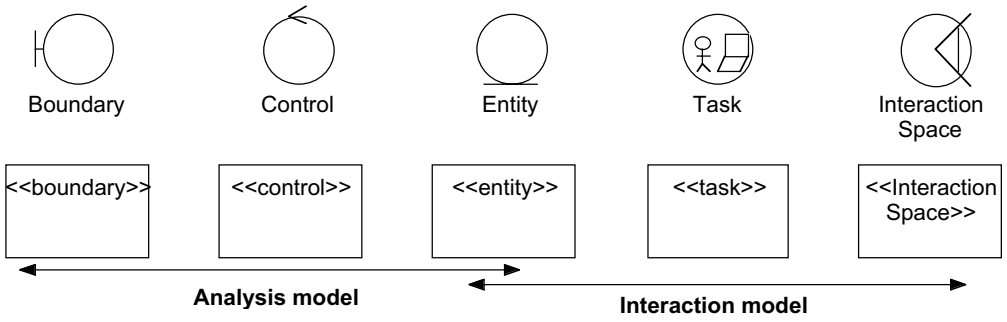


Figure IV.15 – Alternative notations for the class stereotypes of the Wisdom analysis and interaction models.

IV.4.4.UML Extensions for the Dialogue Model

As we discussed in section II.7 task models are fundamental models for user-centered development. In Wisdom essential task flows are used to describe the sequence of activities that a user is required, or believes to be necessary, to do in order to accomplish a goal (see section IV.2.1.2). Therefore, essential task flows are a form of task modeling useful for task analysis purposes, that is, to understand the user tasks that drives development. In section IV.2.1.2 we discussed Wisdom essential task flows and how they can be created in participatory sessions with end-users and later translated into more formal UML activity diagrams. However, at later stages of the development lifecycle, designers are required to perform the transition from task to dialogue, in other words, synthesize the actual description of the human-computer dialogue. As we discussed in section II.7.2, the requirements for dialogue modeling are substantially different from the requirements for task modeling.

Although UML behavioral diagrams (in particular sequence and statechart diagrams) can, to some extent, be used for dialogue modeling, there is evidence that those formalisms are not adequate to specify modern modeless interactive applications (see discussion in section II.7.2). Therefore we decided to adapt one of the most widely used 2nd generation task formalism from the usability-engineering field into the UML.

We chose the *ConcurTaskTrees* (CTT) visual task formalism, because it is a popular and easy to use graphical notation that combines hierarchical structuring of concurrent tasks with a rich set of temporal operators. Moreover, and contrary to other 2nd generation task formalisms (Diane+ and LeanCuisine+ - see section II.7.2), the formal semantics of the CTT are defined and tool support is publicity available.

According to Paternò, the main purpose of CTT is to support the specification of flexible and expressive task models that can be easily interpreted even by people without formal background [Paternò, 2000]. CTT is an expressive, compact, understandable and flexible notation representing concurrent and interactive activities by task decomposition and supporting cooperation between multiple users. CTT defines three types of task allocations:

- user tasks (tasks performed by the user);
- application tasks (tasks completely executed by the application);
- interaction tasks (tasks performed by the user interacting with the system);
- abstract tasks (tasks which require complex activities whose performance cannot be univocally allocated).

An important feature of the CTT notation, essential to bring detail into the dialogue model, is the ability to express temporal relationships between tasks. In the adaptation of the formalism we use UML constraints and stereotyped dependencies to express the temporal relationships between tasks. A constraint is a semantic relationship among model elements that specifies conditions and propositions that must be maintained as true [OMG, 1999]. A UML dependency is a relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element).

The temporal relationships in CTT adapted in our approach are depict in Figure IV.16 and described bellow (including the original notation from [Paternò, 2000] in parenthesis next to their name):

- Independent concurrency (T1|||T2) – denotes that actions belonging to two tasks (T1 and T2) can be performed in any order without any specific constraint;
- Choice (T1[]T2) – denotes that it is possible to choose form a set of tasks and once the choice has been made the chosen task can be performed, while other tasks are not available;
- Concurrency with Information Exchange (T1|[]|T2) – same has independent concurrency but the tasks have to synchronize in order to exchange information;
- Deactivation (T1[>T2) – denotes that the first task (T1) is definitely deactivated once the second task (T2) terminates;
- Enabling (T1>>T2) – denotes that the second task (T2) is activated once the first task (T1) terminates;

- Iteration (T*) – denotes the task (T) is performed repeatedly until the task is deactivated by another task;
- Finite Iteration(s) (T1(n)) – same as iteration but the task (T) is performed n times;
- Optional Tasks ([T]) – denotes that the performance of a task is optional.

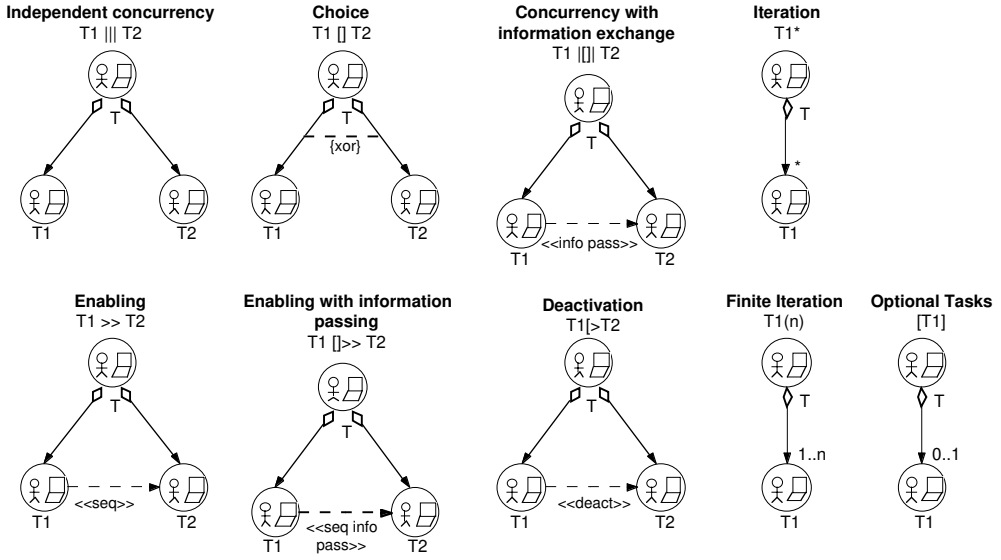


Figure IV.16 – Notation for the UML adaptation of *ConcurTasktrees*

To support the CTT notation the following extensions to the UML are proposed:

- <<Refine task>> is an association between two tasks denoting that the target class (subtask) specifies the source task (parent task) at a different (lower) level of detail. The refine task association is unidirectional can only be used between task classes.

Constraints for the Refine task association (assuming all objects or classes maintain their own thread of control, i.e., they are active and run concurrently with other active objects):

- {xor} – is the UML standard constraint and applies to a set of associations, specifying that over that set, exactly one is manifest for each associated instance;

Dependency stereotypes for the temporal relationships, again assuming that all objects or classes are active and run concurrently with other active objects:

- <<infopass>> is a dependency stereotype denoting the independent task sends information to the dependent task;
- <<seq>> is a dependency stereotype denoting that the dependent task is activated once the first task terminates;
- <<seqi>> is a dependency stereotype denoting that the independent task activates the dependent task with information exchange;
- <<deact>> is a dependency stereotype that denoting that the dependent task is definitely deactivated once the dependent task terminates.

Others have proposed different approaches to integrate task modeling into the UML. As we discussed in [Nunes and Cunha, 2001a] (see also section III.4.3.2 for a similar discussion for task analysis) those approaches can be generally described into:

- Extending the UML metamodel, introducing a separate task model, and establishing relationships between the task elements and the existing UML elements. This approach was the one followed in the CHI'98 workshop (see section III.3.1) but involves a number of problems. On the one hand, the UML metamodel is not yet entirely compliant with the OMG's MOF architecture [OMG, 1999]; hence this kind of heavyweight extension mechanism will likely introduce more inconsistencies in the existing UML semantics. Furthermore it is not possible, in the current UML standard, to devise mappings between the new extended semantics and the notations. Finally, heavyweight extensions require tools that support metamodeling facilities and also impact the interchange formats [Cook, 2000];
- Representing elements and operators of a task model by a UML behavioral diagram (for instance statechart diagrams or sequence diagrams). This approach, proposed in [Markopoulos and Marijnissen, 2000], follows the model-based tradition and suffers from the problems already identified in section IV.2.1.2. In [Silva and Paton, 2000] some of the problems representing task models with statecharts are solved through a set of UML metamodeling extensions, however this possibility incurs into the same problems of heavyweight extensions described earlier.

In [Paternò, 2001] the author agrees that the solution provided for the Wisdom dialogue model enforces the structural correctness of CTT models. However, he argues that the resulting representations are not very effective to support designers in their work: "the usability aspect is not only important for the final application but also for the representations used in the design process" [Paternò, 2001]. We recognize this problem, in particular because the existing UML tools are not specifically adapted to build hierarchical structures, and also because they don't explore the intrinsic possibilities of dialogue notations (for instance simulation). In section V.3 we discuss several experiences were CTT models are interchanged between the CTT environment and the UML through the XMI interchange format. This way we are able to take advantage of specific tool support for dialogue models, while also leveraging the possibility of combining dialogue model with the other modeling constructs required to develop software intensive systems.

IV.4.5.UML Extensions for the Presentation Model

The other design level model defining the Wisdom profile is the presentation model. As we discussed in section III.3.4, the concept of a presentation modeling element to convey the presentational aspects of interactive applications, is commonly accepted in UC-OO methods.

In the Ovid method (see section III.4.2.1), Roberts and colleagues proposed the concept of a (object) view – the modeling concept responsible for presenting to the users and allowing them to use information to accomplish tasks [Roberts et al., 1998]. However the initial proposal of Ovid relies on UML annotations to express relevant information of views and, thus, have little semantic significance. Ovid organizes views in designers' models and implementation models through class, sequence and statechart diagrams. This model organization maps the well-known usability engineering distinction between the different conceptual models of users', designers' and implementation – the aim is to bring the implementation model as close to the user's model as possible, hence, preventing the user interface from reflecting the internal system architecture.

Idiom (see section III.4.2.2) also provides a concept (called view) similar to a Wisdom interaction space. A view in Idiom is described by a type and is an abstract representation of composition of one or more of view objects, interaction techniques and other views. A view can provide a context for task execution by satisfying the user's task information needs and providing ways to invoke application functionality that supports the user's task behavior [Harmelen, 2001b]. The view structure is modeled in the structural view model, and the view behavior supporting task execution is described, including the creation and deletion of top-level views, in the dynamic view model [Harmelen, 2001b].

A similar concept is also proposed in Usage-centered Design (see section III.4.2.3), the authors of essential use-cases propose interaction contexts as the main building block of the presentation aspects of the user interface. An interaction context represents the places within the user interface of a system where the user interacts with all the functions, containers and information needed for carrying out some particular tasks or set of interrelated tasks [Constantine and Lockwood, 1999]. However the authors don't provide any information regarding the definition of interaction contexts within the UML framework. Contrasting Ovid, Usage-centered Design emphasizes the navigational structure of the user interface architecture. Interaction contexts are organized in navigation maps and relationships between interaction contexts are defined (context changes). This characteristic enhances the capability of reasoning about the navigational structure of the interactive application, supporting one of the important compromises in interaction design – the balance between the number of interaction contexts and the number of context transitions.

Both Ovid, Idiom, and Usage-centered Design classify the presentation objects in terms of generic graphical interaction elements, that is, contents, composed, properties and user assistance in Ovid; windows, dialogue boxes, confirmation boxes and palettes in Idiom; and window, screen, display, message and panel in Usage-centered Design. This classification scheme imposes a dependency on the user interface technology - the WIMP graphical user interface, which restricts the ability of the modeling concepts to serve multiple interaction styles and interface technologies.

Interaction spaces in the Wisdom notation have similarities with the concept of boundary classes in UP and the UML standard profile for software development processes, object Views in Ovid, and Interaction Contexts in Usage-centered Design. Figure IV.17 illustrates the main differences between the presentation modeling concepts in UP, OVID, Usage Centered Design, and Wisdom.

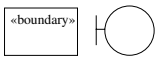
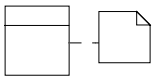
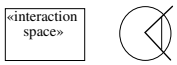
Concept	**UP** Boundary [Jacobson et al., 1999]	**Ovid** View [Roberts et al., 1998]	**Idiom** View [Harmelen, 2001b]	**Usage-centered Design** Interaction Context [Constantine and Lockwood, 1999]	**Wisdom** Interaction space [Nunes and Cunha, 2000b]
Notation					
Definition	Models the parts of the system that depend on its actors (users and external systems).	Presents information to users and allows them to use information to accomplish desired tasks.	Provides the context for task execution, and is an abstract representation of composition of one or more of view objects, interaction techniques and other views.	Represents the places within the UI of a system where the user interacts with all the functions, containers and information needed for carrying out some particular tasks or set of interrelated tasks.	Responsible for the physical interaction with the user, including a set of interaction techniques that define the image of the system (output) and the handling of events produced by the user (input).
Classification	Abstractions of windows, forms, panes, communication interfaces, printer interfaces, sensors, terminals, and APIs.	Classified in composed, contents, properties, and user assistance (help).	Classified in windows, dialogue boxes, confirmation boxes and palletes.	Classified in any, screen or display, window, dialogue or message, panel, or page within tabbed or compound dialogue.	No specific classification. Stereotyped attributes (input and output) elements, operations (actions), and associations (navigational and containment).
Organization	Organized in the analysis model in class, sequence, collaboration, state-chart and activity diagrams.	Organized in designers and implementation models in class, sequence, and state-chart diagrams.	Organized in view structural models.	Organized in navigation maps.	Organized in the presentation model in class and activity diagrams.

Figure IV.17 – Presentation modeling elements in the UP (boundary), Ovid (object view), Idiom (view), Usage-centered Design (interaction space) and Wisdom (interaction space)

In the Wisdom notation the presentation model defines the physical realization of the perceivable part of the interactive system (the presentation), focusing on how the different presentation entities are structured to realize the physical interaction with the user. The presentation model provides a set of implementation independent modeling constructs (the <<interaction space>> class stereotype) for use by the presentation model, hence, leveraging independence of the interaction techniques provided by the user interface technology (e.g. UI toolkit). Interaction spaces are

responsible for receiving and presenting information to the users supporting their task. Views are typically organized in hierarchies and containment relationships can occur between views. In the next section we present the UML's extensions to support the presentation model.

The following UML extensions support the Interaction model:

- <<Navigate>> is an association stereotype between two interaction classes denoting a user moving from one interaction space to another. The navigate association can be unidirectional or bi-directional; the later usually meaning there is an implied return in the navigation. Users navigate in interaction spaces while performing complex tasks and a change between interaction spaces usually requires a switch of thinking from the user;
- <<Contains>> is an association stereotype between two interaction space classes denoting that the source class (container) contains the target class (content). The contains association can only be used between interaction space classes and is unidirectional.
- <<input element>> is an attribute stereotype denoting information received from the user, i.e., information the user can manipulate;
- <<output element>> is an attribute stereotype denoting information presented to the user, i.e., information the user can perceive but not manipulate;
- <<action>> is an operation stereotype denoting something a user can do in the physical user interface that causes a significant change in the internal state of the system, that is, changes in the long term information of the system (entities), request for signification functionality, changes in context of the user interface, and so on.

IV.4.6.Valid Association Stereotypes Combinations

Figure IV.18 illustrates the valid combinations for the different associations stereotypes defined in the UML extensions of the Wisdom profile.

To: From:	Human actor	System actor	Boundary	Control	Entity	Task	Interaction Space
Human actor	resembles specializes includes						communicate
System Actor			communicate				
Boundary		communicate	communicate	communicate	communicate subscribe		
Control			communicate	communicate	communicate subscribe	communicate	
Entity					communicate subscribe		
Task				communicate	communicate subscribe	communicate refine task	
Interaction Space						communicate	navigate contain

Figure IV.18 – Valid association stereotypes combinations

The previous sections described the main contributions underlying the Wisdom method: process, architectural models and notation. The following section describes how those contributions are combined together supporting interactive system development with the Wisdom method.

IV.5. THE WISDOM METHOD

This section presents the Wisdom method in detail, introducing the specific activities involved in each of the three major workflows for software development: requirements, analysis and design. In the following subsections we describe the activities for each workflow, present the main techniques used in Wisdom and illustrate the application of the method with a simple hotel reservation system.

IV.5.1.Genealogy of Wisdom

For several years we worked with small software developing companies (SSDs) to try to help them build a rational software development process. We have learnt how people in such environments build interactive software, understood the methods, techniques and tools they use, characterized the kind of products they have built, and studied how they have worked together and with their customers. The Wisdom proposal results from our efforts to bring rational software development practices into these environments.

Wisdom originated from a simpler 1997 proposal called User-Centered Evolutionary Prototyping (UCEP) [Nunes, 1997; Nunes and Cunha, 1998; Nunes et al., 1998]. UCEP was a customized set of methods and techniques aiming to rationalize a chaotic software process in a particular small software developing company in the Island of Madeira, Portugal. Our initial approach was applied to a medium scale project, involving a small team of 6 people. The project saw the development of a supervisory and decision support system (SITINA) for the power management company in Madeira [Nunes et al., 1998]. The UCEP set of methods, tools and techniques we adapted and transferred into the software developing company were:

- object-oriented methods - at the time an adapted version of the Object Modeling Technique (OMT) [Rumbaugh et al., 1991];
- visual modeling tools - initially Rational Rose and in the end a simple drawing package due to integration and platform restrictions;
- low-fi and high-fi prototyping [Isensee and Rudd, 1996]; and finally
- participatory techniques - used to support requirements discovery, introduce notation and evaluate prototypes.

Figure IV.19 illustrates the Wisdom genealogy; on the left hand side of the figure are major software engineering influences, on the right hand side major human-computer interaction influences. From the illustration we can see that Wisdom was inspired in Bohem's Spiral Model [Bohem, 1988] and Kreitzberg's LUCID [Kreitzberg, 1996].

From the spiral model Wisdom got the evolutionary rhythm, and from LUCID the user-centered perspective. Since its initial form, the Wisdom approach used an object-oriented analysis and design notation to specify and document the software system. We chose OMT because of it's easy to learn notation and simple transformation rules for the relational schema. The OMT notation (essentially the static model) played two major roles in UCEP On the one hand it worked as the common language for practitioners, and between practitioners and users in participatory sessions and prototype evaluation. On the other hand, the working models stabilized the evolutionary process and documented the projects.

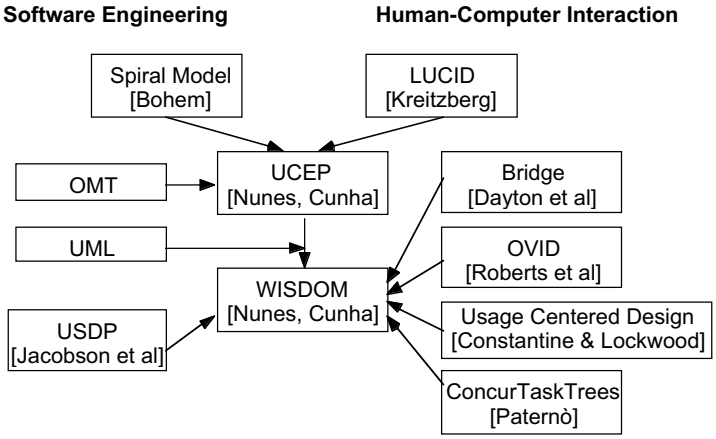


Figure IV.19 – Wisdom Genealogy

The success of several following experiments with UCEP, and the advent of the UML led to the proposal described here and renamed Wisdom. The UML provided new modeling constructs and built-in extension mechanisms, essential to formalize some off-hand modeling techniques used in UCEP, like the ones that specify the user interface and essential task flows. Special care was also taken to improve UCEP's participatory techniques. Participatory techniques play a major role in Wisdom because they are used to bring users to the development process, to introduce the notation on a need-to-know basis and to take advantage of SSD's enhanced communication capabilities. To support participatory requirements gathering and participatory design, Wisdom adapts some ideas from the Bridge method [Dayton et al., 1998]. Since the emphasis was not only to produce object-oriented user interfaces, like in the Bridge method proposes, we decided to detach the technique from this specific interaction style.

Essential use-cases [Constantine, 1992] provided a sound and important tool for task modeling based on use-cases, which is the de facto standard in OO development. They also provided an effective connection to user profiling model-based techniques such as the ones promoted with user role modeling in Usage-centered Design [Constantine and Lockwood, 1999]. Therefore, Usage-centered Design is a major influence for modeling techniques in early development activities. The importance of a conceptual construct to capture the presentation aspects for user interface design, proposed in early UC-OO methods like OVID and Idiom, influenced the presentation

model in Wisdom. To specify the dialogue model we adapted the *ConcurTaskTrees* [Paternò, 2000] task formalism, hence providing support for reusable task structures, task patterns and task-based evaluation of the user-interface. Finally we decided to use the software process modeling terminology of the UP framework [Jacobson et al., 1999] in Wisdom. Being also focused on improvement strategies for SSDs, we required a terminology that clearly mapped to the existing best practices on the object-oriented software engineering field.

Due to the degree of customization to integrate human-computer interaction techniques, Wisdom is not intended to develop non-interactive software. Wisdom was not tried on large-scale projects, as the remit is not intended for such projects and large development teams. Furthermore, Wisdom was only applied on custom or in-house projects. Wisdom was never tried in off-the-shelf and contract based development contexts [Grudin, 1991]. Although Wisdom is not intended to be a scalable method, we believe it can be applied within other contexts and represent a good step towards a large-scale industry process like the Rational Unified Process (RUP) [Kruchten, 1998] or Catalysis [DSouza and Wills, 1999]. Wisdom is strongly based on the UML and it uses several models and views that exist in such large industrial processes. Thus, an organization can start implementing Wisdom and then incrementally introduce new workflows, models and views using the UML, to then ultimately reach an industry scale process.

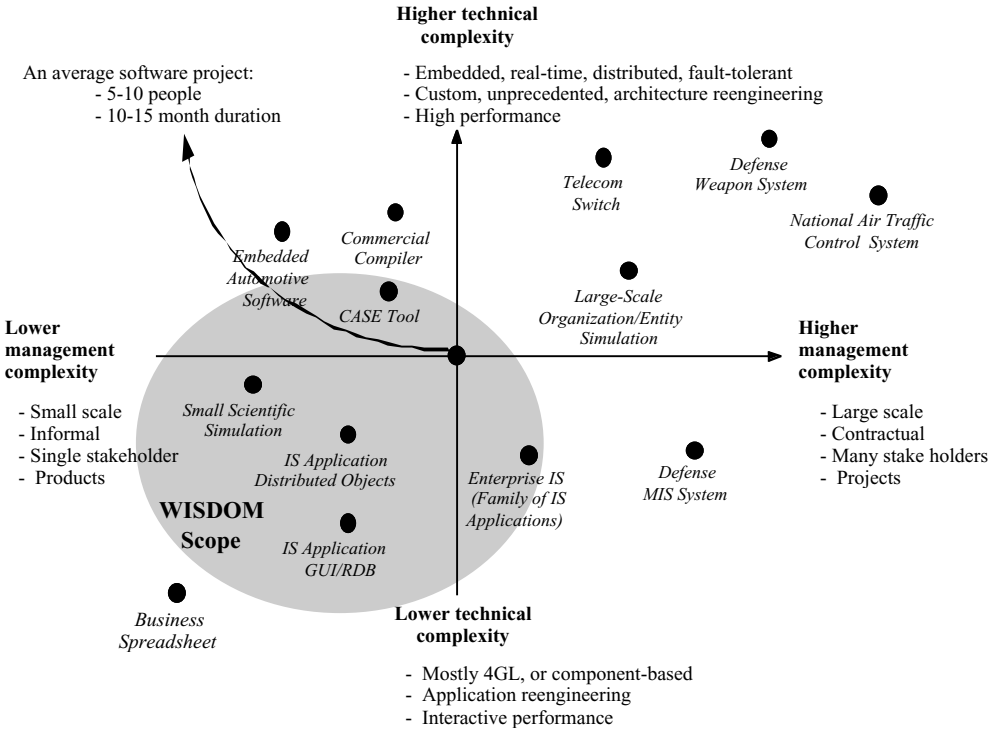


Figure IV.20 – Dimensions of Software Complexity and Wisdom (adapted from [Royce, 1998])

Wisdom is a lightweight software engineering method. Lightweight in the sense that it can be learnt and applied in a couple of days or weeks. Wisdom is object-oriented, it uses the UML to specify, visualize and document the artifacts of the development

project. Wisdom is specifically adapted to develop interactive systems, because it uses and extends the UML to support human-computer interaction techniques. Finally Wisdom is evolutionary, in the sense that the project evolves in a sequence of incremental prototypes, ultimately leading to the end product.

IV.5.2.Workflows, Development Activities and Models in Wisdom

The three Wisdom software development workflows presented in this section are depicted as UML activity diagrams in Figure IV.21, Figure IV.23 and Figure IV.25. Despite the problems representing evolutionary processes with sequential notations, we recognize the importance of such representation to clearly present the method. To overcome some of those problems we introduce several graphical notations to better express Wisdom's whitewater evolutionary nature (see section IV.2). Each graph represents the Wisdom activities for each *conventional* software development workflow (requirements, analysis, design and whitewater evolution). The Wisdom specific activities, introduced in section IV.2, are illustrated with three cogwheels (stereotypes of activities that denote software development activities). Sub-activities are illustrated as two cogwheels and, where required, grouped in a rounded rectangle to detail a specific Wisdom activity. Due to the whitewater nature of Wisdom several activities can occur in parallel: a double sized line with multiple outgoing arrows illustrates such parallel execution (fork and join). Output artifacts of the Wisdom activities are illustrated with corresponding notational icons and related to the activities with a dashed dependency. We recognize this representation to be non-compliant to the UML specification, but use it because we believe it's clearer.

We claim that the activity diagrams in Figure IV.21, Figure IV.23 and Figure IV.25 represent a *hypothetical* implementation of the Wisdom method and should only work as a framework. Depending on the complexity of the projects, the maturity of the development team and other factors, the flow of activities can be different. For instance, some activities can be eliminated in simple projects, while others added for different purposes (e.g. quality procedures, process management). This can happen within teams, projects or even iterations in the same project performed by the same team. To better illustrate this whitewater nature of Wisdom, suppose we draw each activity of a specific workflow in a sheet of paper and throw them into a rapid stream of water (whitewater). What usually happens is that some sheets find obstacles and reach the target later (or never) than others that are able to find a quicker path. If we repeat (iterate) the experience the pace differs each time, and sheets that failed to reach the target (or moved slowly) the first time can perform exceptionally on a different turn. In the end this apparently messy approach is usually better and more flexible than sequentially having the sheets traveling in a canal at the same but slower speed.

As mentioned in the beginning of this section we use a simple hotel reservation system to illustrate our method. The examples shown bellow and throughout this

section are based on a simple problem definition based in similar examples worked in the literature [Roberts et al., 1998; Dayton et al., 1998]. To clarify the scope of the case study we cite a definition of this particular problem from [Nunes et al., 1999].

> *"The guest makes a reservation with the Hotel. The Hotel will take as many reservations as it has rooms available. When a guest arrives, he or she is processed by the registration clerk. The clerk will check the details provided by the guest with those that are already recorded. Sometimes guests do not make a reservation before they arrive. Some guests want to stay in non-smoking rooms. When a guest leaves the Hotel, he or she is again processed by the registration clerk. The clerk checks the details of the staying and prints a bill. The guest pays the bill, leaves the Hotel and the room becomes unoccupied." [Nunes et al., 1999]*

We use this simple example to illustrate the Wisdom method because it is easier to understand that a real life problem. Examples of the applicability of the Wisdom method to real-life problems can be found in [Nunes and Cunha, 2001b], where we present a case study with different Wisdom artifacts produced for a web-application developed to support cooperative work in the construction industry.

IV.5.3.Requirements Workflow

The purpose of the requirements workflow is to aim development towards a system that satisfies the customer (including the end users). At this level Wisdom differs from the mainstream object-oriented software engineering process frameworks introducing several activities to support the participatory user-centered perspective. Figure IV.21 represents an UML activity diagram describing the main activities in the requirements workflow.

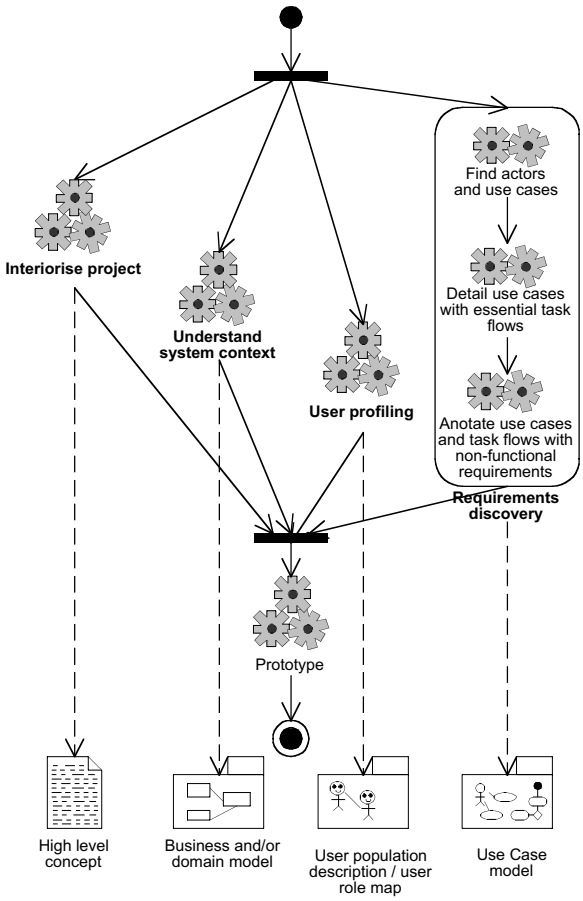


Figure IV.21 – Activity Diagram for the Requirements Workflow Set in Wisdom.

The **interiorize project** activity (leftmost in Figure IV.21) aims at raising some initial brainstorm over the scope of the envisioned system. In this activity the end-users and the team devise a high-level concept of the problem. This short textual description, usually a single paragraph [Kreitzberg, 1996], should clearly indicate what the system should and should not do, and what are the potential benefits and anticipated risks. One can see this description as an initial user manual of the system (for more information on this approach see [Norman, 1998]).

The **understand system context** activity aims at understanding the domain of the problem, either focusing only on the *things* involved in the problem domain, or, also in the users and their work processes. Usually the former simplified version happens when the problem domain is very simple or the development team is experienced in the domain. The resulting artifact is then a domain model of the problem, usually an UML class diagram describing the business or real-world objects manipulated in the problem domain and their relationships. The later, more complex version, happens when the problem domain is complex or when the development team has little or no knowledge of the domain – very frequent in SSDs due to their flexibility. In such case the development team produces a business model, describing the workers, the work processes they perform and the business entities they manipulate. The outcome is again an UML class diagram, using the business process profile of the UML, and

several activity diagrams describing the business processes. It is out of the scope of this thesis to discuss this standard profile (refer to [OMG, 1999] for further details).

The purpose of **user profiling** activity is to describe (profile) the actual users whose work will be supported by the system. Here, the goal is to describe who are the prospective users, how they are grouped and what are their salient characteristics (incumbents, proficiency, interaction, information, usability criteria, functional support and other criteria as described in section IV.4.1). Depending on the complexity of the project the outcome of this workflow can be a simple textual description of the users or a complete user role model as described in section IV.4.1.

The **requirements discovery** activity aims at finding and capturing the functional and non-functional requirements of the envisioned system. Wisdom is a essential use-case and task flow driven method; therefore it relies on essential use-cases [Constantine and Lockwood, 1999] to capture the structure of use and the underlying functional requirements. Because of the evolutionary nature of the method, non-functional requirements can largely influence the quality and stability of the end product. That way we propose to attach non-functional requirements to the use-case model at different levels of abstraction. Thus, the requirements discovery activity encompasses several sub-activities (at the far right in Figure IV.21), they are (i) finding actors and essential use-cases; (ii) detailing essential use-cases with activity diagrams; (iii) annotating non-functional requirements to use-cases. At this level there are several differences between Wisdom and the UP approaches. Our approach uses activity diagrams to detail essential use-cases, expressing the desirable and realistic task flows. These task flows drive the user interface design process ensuring that the interaction model reflects the actual perceived tasks of the users. The Wisdom approach also encourages the use of annotations to express non-functional requirements in the actual use-case model. This technique reduces the number of artifacts the development team must master and maintain. Also, non-functional requirements can be placed at the hierarchical level best suited to them, i.e., annotating an activity in the activity diagram, annotating a whole activity diagram, or even annotating the entire use-cases model for high level non-functional requirements.

Usually the Wisdom evolutionary prototyping approach ends the requirements workflow with one or several prototypes. At this stage prototypes are typically low-fidelity and non-functional (see section IV.2.1). However, functional prototypes are sometimes useful to illustrate a novel technology and jumpstart requirements discovery. We have witnessed several successful experiences where a functional prototype is useful to demonstrate to the clients the potential of a given novel solution, fostering a better understanding of the impact of the envisioned system.

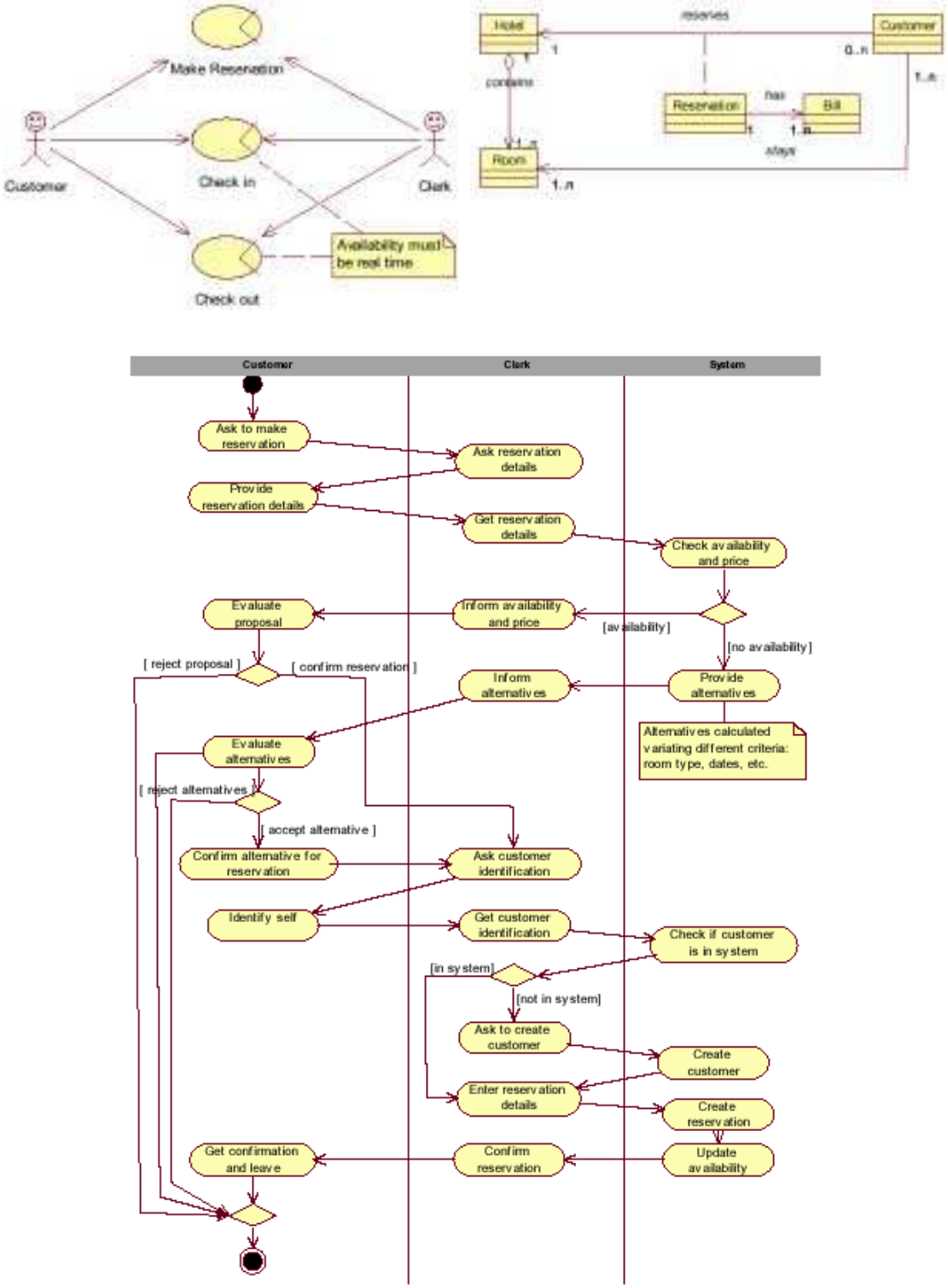


Figure IV.22 – Example of artifacts produced by the Requirements Workflow for the Hotel Reservation System

Figure IV.22 represents three artifacts from the requirements workflow. On the top left-hand side of the figure is the use-case model annotated with one non-functional requirement. To the right is an essential task flow expressing the make reservation use-case. The first two swim lanes illustrate which actor performs what activity; the last swim lane depicts system responsibilities. This example of an essential task flow

includes detailed business information (the workflow between customer and clerk) and is a product of several iterations and participatory sessions with end users. The aim is to exemplify the informing power of early models like the business model. Finally in the bottom left-hand side is a domain model showing the most important *things* in the problem domain and how they relate to each other. Just as a simple example of the importance of participatory techniques to create requirements models, we asked several experienced practitioners to build a domain model from the above problem description. We also worked the example in a short participatory session with domain experts. The former considered the hotel the universe of discourse and omitted the class from the domain model. The later included the hotel class because it is clearly a user perceivable object, and, therefore, important from the user interface perspective.

IV.5.4.Analysis Workflow

The analysis workflow refines and structures the requirements described in the previous workflow. The purpose is to build a description of the requirements that shapes the structure of the interactive system. Whereas the use-case model is an external view of the system, described in the language of the users; the analysis model is an internal view of the system in the language of the developers.

The activity diagram for the Wisdom analysis workflow is illustrated in Figure IV.23. The major activities in this workflow reflect the incremental construction of structured stereotypical classes and package diagrams. The separation of concerns between the internal architecture and the user interface architecture is responsible for the two concurrent flows of activities in the illustration. The leftmost flow reflects the building of the internal architecture of the system and how analysis classes are organized to realize the different use-cases. The rightmost flow structures the interaction model in terms of task and interaction space classes and how they are organized to support the user interface of the system. Note that the rightmost activity is itself divided in two concurrent flows that support the separation between the dialogue and presentation structures of the interactive system (see section IV.3.2).

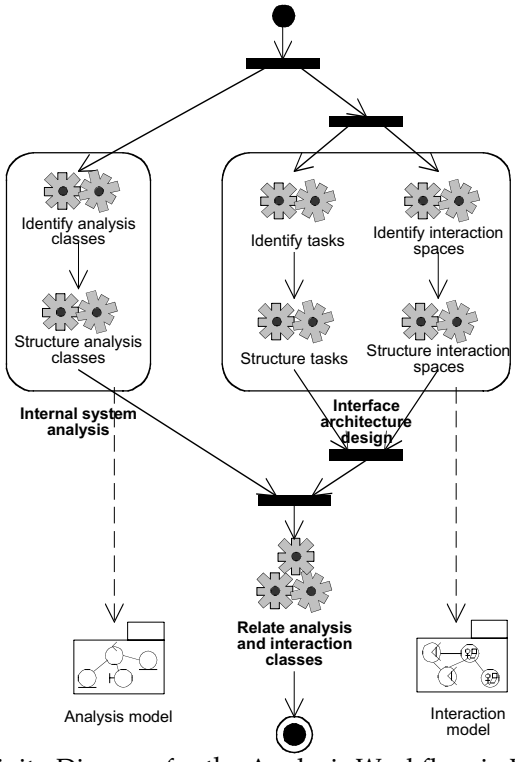


Figure IV.23 – Activity Diagram for the Analysis Workflow in Wisdom

The **internal system analysis** activity aims at structuring the requirements in a way that facilitates their understanding and management. Analysis classes represent abstractions of domain concepts captured in the requirements workflow, and are focused on functional requirements, postponing the handling of non-functional requirements to design. They contain attributes, operations and relationships at a high level of abstraction, i.e., attributes are conceptually typed, operations are informal descriptions of responsibilities and relationships are more conceptual than their design and implementation counterparts. There are two sub-activities in this internal system analysis: identify general analysis classes and structure analysis classes. There are several techniques to extract analysis objects from the requirements model. Wisdom does not define any special technique to do this; instead we recommend a lightweight version of the CRC card method [Beck and Cunningham, 1989]. We found out that this method, for its close resemblance to the participatory nature of the Bridge method (see section III.4.2.4), can be easily introduced as an effective way to extract analysis classes and corresponding responsibilities from the requirements model. The next sub-activity is to structure the identified classes into analysis stereotypes and distribute responsibilities to build the internal architecture. This is usually a sub-activity that takes several iterations, prototypes and visits to other activities in the requirements workflow until a stable architecture is reached.

The **interface architecture design activity** concerns the external architecture of the system, i.e., the part of the system responsible for the user interface supporting the users performing the envisioned tasks. As mentioned before this activity compromises two concurrent flows of activities corresponding to the dialogue and presentation

models of the user interface. Like in the internal architecture companion activity, those two flows of activities have also two sub-activities: identify and structure interaction classes (task and interaction space). As we discussed in section IV.3.2 Wisdom defines two stereotypes for interaction classes. The task class stereotype is the element of the dialogue component, and the interaction space class stereotype is the element of the presentation component. While devising the user-interface architecture we should concentrate on the desirable tasks and not in a particular user interface design solution. That is the role of the design level models (presentation model and dialogue model) in the design workflow. To identify both the task and interaction space classes we use the essential task flows from the requirements workflow as the main source of information. Tasks have usually a clear mapping to the task steps in the task flows. The major problem at this level is to identify a set of tasks that support the essential task flows while ensuring there is robustness and reuse in the user-interface architecture. Regarding the presentation component, identifying and structuring interaction spaces is mainly a compromise between the number of interaction spaces and the number of transitions between interaction spaces. This is usually done producing several versions of the architecture and testing different representations through abstract prototypes in usability sessions with end users (in [Dayton et al., 1998] methods are provided to perform usability evaluation within the Bridge method). Like in the previous activity the user-interface architecture usually takes some visits to the requirement workflow before a stable architecture can be achieved.

The final activity in the analysis workflow is to **relate the internal and user-interface architectures** to see how they collaborate in realizing the essential use-cases. This activity is very important to balance both architectures and ensure the design and implementation models can seamlessly build up on them. This sub-activity is done creating associations between classes in the two models, usually communicates or subscribes association stereotypes between entities in the analysis model and tasks in the interaction model (see section IV.4.3).

If required, there is an additional activity in this workflow. This optional step is to package analysis classes and interaction classes into smaller manageable pieces. Since use-cases capture task flows and functional requirements, they are the natural basis for packaging. Therefore, packaging should allocate groups of use-cases, preferably related in the same domain or business model, to analysis packages. These packages will then fulfill the use-cases user-interface and underlying functionality accordingly.

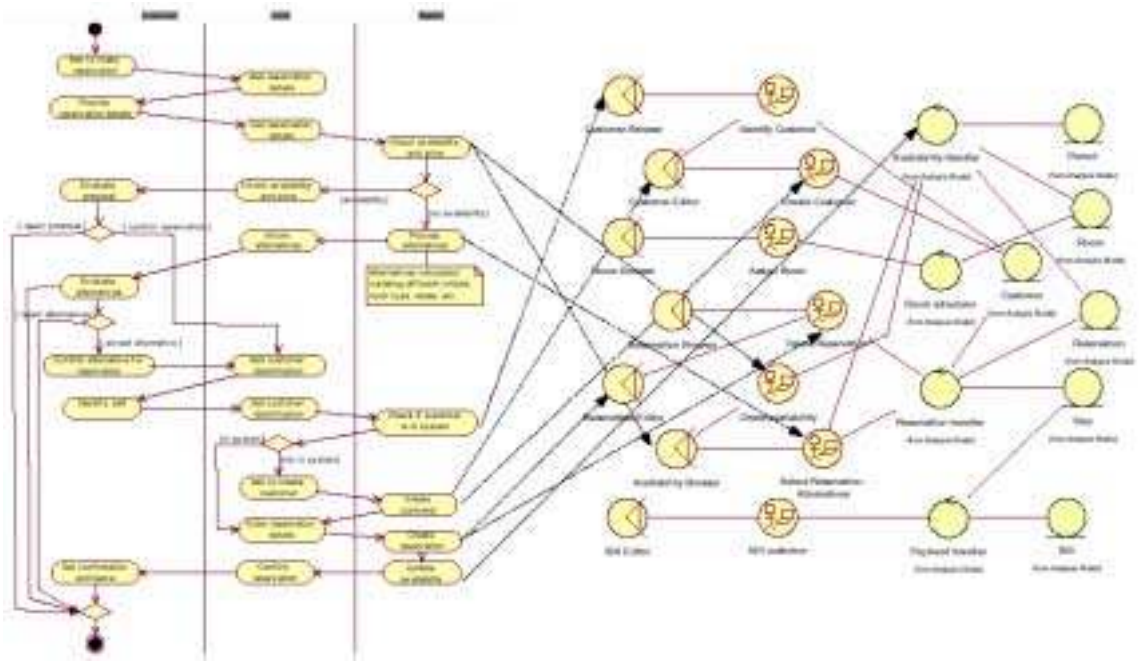


Figure IV.24 – Example of artifacts produced by the Analysis Workflow for the Hotel Reservation System

Figure IV.24 illustrates three artifacts from the analysis workflow for the Hotel reservation system. At the left hand side is an essential task flows (activity diagram) expressing the *make reservation* essential use-case. To the right of the activity diagram is a possible interaction model with stereotyped interaction classes (interaction spaces and tasks) and corresponding associations. The arrows between those two diagrams represent mappings used to identify interaction spaces and tasks from the essential task flow. As we can see, the mapping is simple and straightforward. At the right hand side is a possible internal architecture model with stereotyped analysis classes (control and entity) and corresponding associations (communicates). The relationship between tasks classes and entity or control classes ties both architectural models together and corresponds to the last but one activity in Figure IV.23 (relate analysis and interaction classes). A comparison between this analysis model and the domain model in Figure IV.22 clearly points out the goal of the internal analysis activity. The former model is model of the problem domain while the later an architecture of a solution. Although this is a simple example, note that not all the entities identified in the analysis model are user perceivable objects (objects of interest to the users). This example also depicts multiple analysis and interaction classes collaborating in the realization of one or more essential use-cases. For instance, *the customer browser* interaction space, the *identify customer* task, the *customer* entity and the *availability handler* control; all amongst other realize the make reservation essential use-case. In addition they all, except the *availability handler* control, realize also the *check in* and *check out* use-cases (see Figure IV.22).

IV.5.5.Design Workflow

The design workflow drives the system to implementation refining its shape and architecture. While the analysis workflow only focuses on functional requirements, the design workflow also focuses on non-functional requirements. At this level constraints related to the development environment (languages, databases, GUIs, operating systems, distribution, replication, etc.) are managed. During design the system is then broken up into manageable pieces, possibly allocated to different development teams (concurrently or not).

The activity diagram for the design workflow is illustrated in Figure IV.25. The major activities in this workflow reflect the incremental refinement of the analysis and interaction models. The separation of concerns initiated in analysis continues during the design workflow. Since the design workflow tends to progress through a succession of incremental prototypes, usually, two activities take place simultaneously: the *internal system design* and the *user interface design*. One of the major improvements of Wisdom, at this stage, is enabling careful planning of design (and consequent implementation) prioritizing the evolutionary process by essential use-cases and interaction classes, i.e., the architectural building blocks. The internal and user interface architectures devised in the previous workflow guarantee that the incremental construction brings *muscle* (design) to a coherent *skeleton* (analysis).

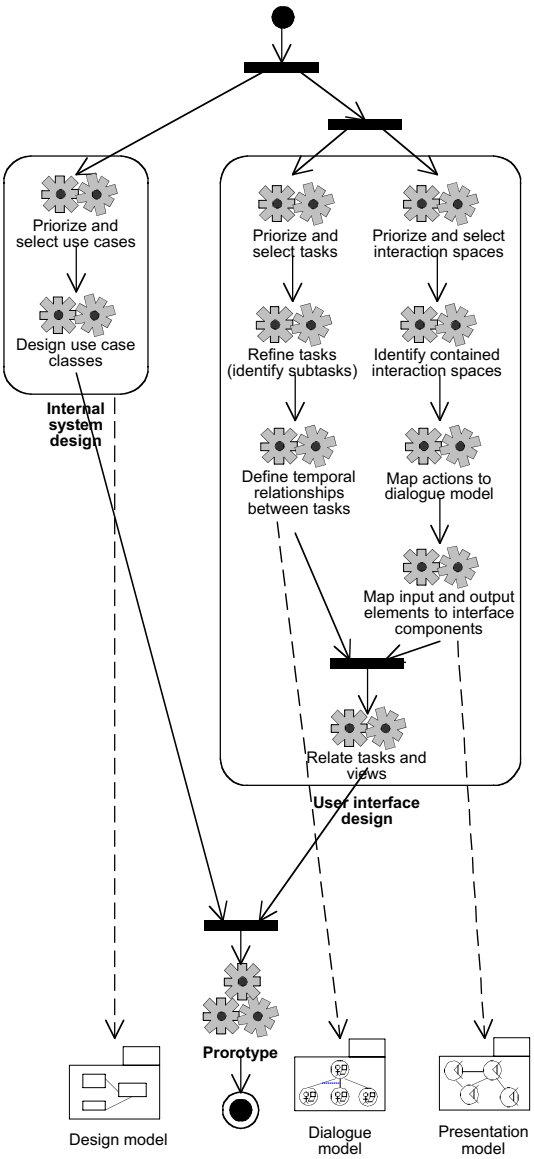


Figure IV.25 - Activity Diagram for the Design Workflow in Wisdom

The **internal system design** activity is illustrated on the left-hand side of Figure IV.25. Since design highly depends on the development environment (tools, technologies, programming languages, etc.) we will only briefly describe its goal. The first sub-activity is to prioritize and select candidate use-cases for design. As we saw in section IV.2.1.2, use-cases drive the development process binding the workflows together. At this stage, information from the use-case model, including the essential task flows, is used to give cadence to the evolutionary process. The unified process places this activity in the early phases of the development process; in Wisdom such decision is done in design, as experience has shown that SSDs tend to move faster in the early phases of development. Prioritizing decisions is often achieved well after an in-depth understanding of the system under development. The following design activity in internal system design accomplishes the refinement of the stereotypical analysis classes, both at the responsibility and association level. Refinement is achieved once the integration of the non-functional requirements has taken place. The importance of annotating non-functional requirements to the requirements model is ultimately

important at this stage. Since they escort the use-cases that drive the development process, the risk of leaving out such requirements is minimized and leverages better design decisions. This refinement process, in SSDs' environments, also leads to a partial translation of the analysis model (typically entity stereotypes) to the relational schema. This translation process between entity classes and the relation schema is widely documented in [Blaha and Premerlani, 1997]. Moreover there is increasing interest in standardizing a UML profile for data modeling - at least one mainstream UML tool (Rational Rose) already provides support for such a profile and partially automates the translation process [Rational, 2000]. Nevertheless, the characteristics of the 4GLs used, increase the risk of driving the user interface from the database schema. Thus, the increased importance to separate the user interface from the design model.

The **user-interface design** activity is concerned with the realization of the interaction model into a concrete user interface. This activity usually encompasses two main design decisions or restrictions: the interface style and the user interface technology (see section III.1). They both influence each other, e.g., a particular user interface technology can prevent the implementation of a specific interaction style. The construction of the presentation model, detailed at the right hand side of Figure IV.25, encompasses two concurrent flows of activities, corresponding to the dialogue and presentation components of the interaction model. The leftmost flow of activities corresponds to the refinement of the tasks identified and structured in the analysis workflow. This process usually involves three steps: (i) decide which tasks to refine, this is usually done in co-ordination with the prioritization used for internal functionality; (ii) identify which subtasks refine the top-level task; (iii) define the temporal relationships amongst the subtasks according to the options presented in section IV.4.4. Whenever possible, task patterns should be used in steps (ii) and (iii) [Tidwell, 1998a; Bayle et al., 1998; Welie and Troedtteberg, 2000] (see also section V.4). The rightmost flow of activities corresponds to the refinement of the presentation model defined in the analysis workflow. This process usually involves four steps: (i) decide which interaction space to refine, again this is usually done in co-ordination with the use-case prioritization defined for internal functionality; (ii) decompose complex interaction spaces in different contained or navigable interaction spaces, therefore creating interaction spaces that easily map to the implementation toolkit and also to presentation patterns fostering reuse; (iii) map actions on interaction spaces to the dialogue model, hence establishing an initial correspondence between the dialogue and presentation models; and finally (iv) map input and output elements to interface components, for instance mapping elements to GUI widgets. The user-interface design activity ends relating both task classes to interaction space classes, hence, completing the process of distributing responsibilities between the dialogue and presentation models.

The following criteria have been identified to gather information from the task model that is also useful to guide the user interface design process [Paternò, 2000]:

- Task type, frequency and cognitive effort – the type of presentation depends on the information required to perform the task (the type of task), for instance, if a task has to consider a spatial relationship it is important to provide a graphical representation. The frequency with which a user performs a task, or sequences of related tasks, also influences the presentation, for instance frequent tasks should be highlighted in the presentation and shortcuts provided. Finally the cognitive effort underlying a task can be reduced by synchronizing different media, for instance, it is problematic to ear a long description and read at the same time;
- Contextual environment and user knowledge – we have to take into account the context in which a particular task is performed and also the impact of the user knowledge in the application domain. These issues are discussed in user-centered design in section II.5;
- Layout optimization – the performance of the same task sometimes requires different amounts of information depending on the specific instances of objects involved, thus the structure of the information remains the same but the layout changes to accommodate different information;
- Identifying and group presentation objects (interaction spaces) – the concept of an enabled task set is useful for identifying interaction contexts. An enabled task set is a set of tasks that are logically enabled to start their performance during the same period of time [Paternò, 2000]. One task can belong to multiple enabled task sets, but if multiple tasks are enabled at the same time they should be grouped in the same presentation (interaction context);

One important aspect with identifying interaction contexts is that there is a large set of possibilities in terms of the number of distinct presentations that support a given task or set of interrelated tasks [Paternò, 2000]. Paternò identifies three types of possible approaches [Paternò, 2000]:

- One unique interaction space supporting all the possible tasks – this approach is possible for simple applications, otherwise the resulting interface would be confusing because there is large number of interaction techniques in a very limited space;
- As many presentations as the number of possible enabled task sets – the number of enabled task sets, associated with a task model, is the upper bound to the number of distinct and meaningful interaction spaces. If the number of presentations is higher than the number of enabled task sets then the user interface imposes a sequential order on the performance of some tasks that could be enabled concurrently;
- Intermediate solutions – the intermediate solution implies that tasks belonging to different task sets are supported by the same presentation (interaction space). This can be beneficial when different tasks are performed in sequence many times or when they exchange information, thus are logically related. The criteria for intermediate solutions depends on temporal relationships among tasks, for

instance, tasks that exchange information should be placed in close proximity, if there is a choice among tasks the presentation should highlight the different possibilities, if there is a disabling task then the control task should be placed in a predefined location.

Like the internal system design is highly dependent on the development environment, also the user interface design depends on a large number of external factors. Our goal in Wisdom was to build a model-based framework that could accommodate different design techniques, user interface styles and technologies. On the one hand, we have an UML model based approach that detaches the user interface design techniques from a specific design context, using the abstract concepts of tasks and interaction spaces. On the other hand, those abstract concepts are refined during design time supporting different design alternatives to create the concrete user interface, leveraging multi-user interface design over the same interface architecture (see section V.1.3). The Bridge method [Dayton et al., 1998] is one alternative to design concrete user interfaces in Wisdom, one that clearly fits the SSDs environment. Hence, if the development team decides to design an object-oriented graphical user interface Part 2 and 3 of the Bridge can be easily implemented within Wisdom (see sections III.4.2.4 and III.4.2.4 for a description of the Bridge part 1). Part 2 of the Bridge concerns mapping task flows to task objects; the goal is to map the user interface requirements into discrete units of information that users manipulate to do the task (task objects). In Wisdom the task objects map to interaction spaces. Therefore the specific behaviors and containment relations for task objects in the Bridge are, in fact, Wisdom stereotyped interaction spaces, with input and output elements, actions, navigational and containment associations. In this OO GUI context we can use the Bridge task object design (TOD) technique to discover, design, document and usability test the task objects. Afterwards we can use Part 3 of the Bridge to map task objects to GUI objects. This 3rd step of the Bridge guarantees that the GUI is usable for executing the task. For an in depth description of the Bridge refer to [Dayton et al., 1998]. For a brief illustration of this integration of the Bridge in Wisdom see the example in Figure IV.27. Concluding, a similar distinction between part 1, and parts 2 and 3 of the Bridge, should be taken when using a different design method, style and technology.

Iterations through the Wisdom design workflow usually reflect the actual implementation of the end product. At this level prototypes are usually fully functional incremental evolutions of the end product, typically organized in terms of prioritized use cases and interaction classes. This nature of development is very important for SSDs, since they are very reluctant to throw away functional prototypes. If adequately implemented, Wisdom provides the needed control over the evolutionary process to quickly and effectively produce high-quality products from this final Wisdom workflow.

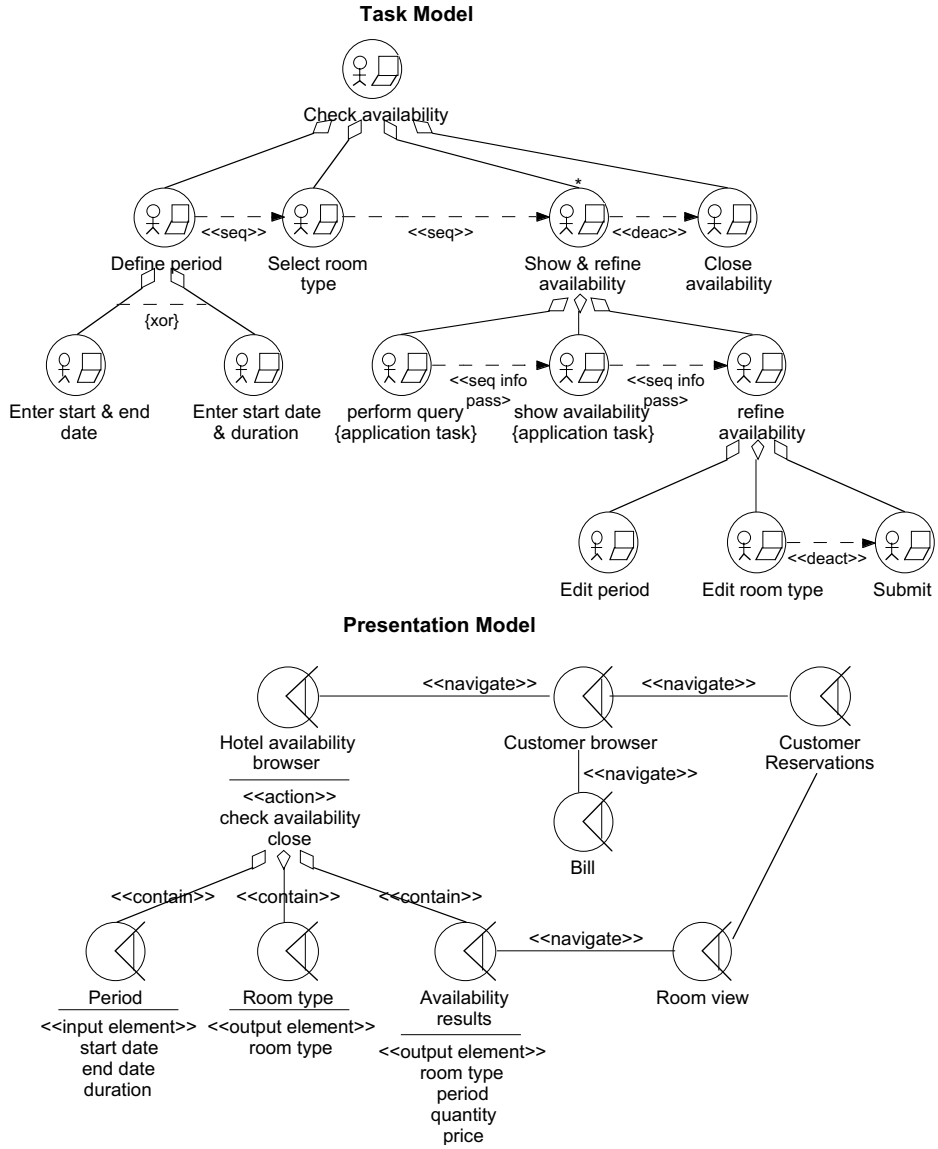


Figure IV.26 - Artifacts from the Design Workflow for the Hotel Reservation System

Figure IV.26 illustrates two Wisdom design artifacts for the hotel reservation system. The top diagram is part of the dialogue model, in particular, the refinement of the top-level task *check availability* (see related example in Figure IV.24). This particular example uses the search task pattern (see [Paternò, 2000] and section V.4). The diagram in the bottom of the figure represents part of the presentation model. This example illustrates two advantages of the presentation model. While the left hand side of the diagram details the presentation structure that supports the top-level task *check availability*, the right hand side shown the top-level navigation structure. That way, developers are able to zoom into detail while keeping focus on the high level structure, thus preventing the well-known problems of incremental approaches.

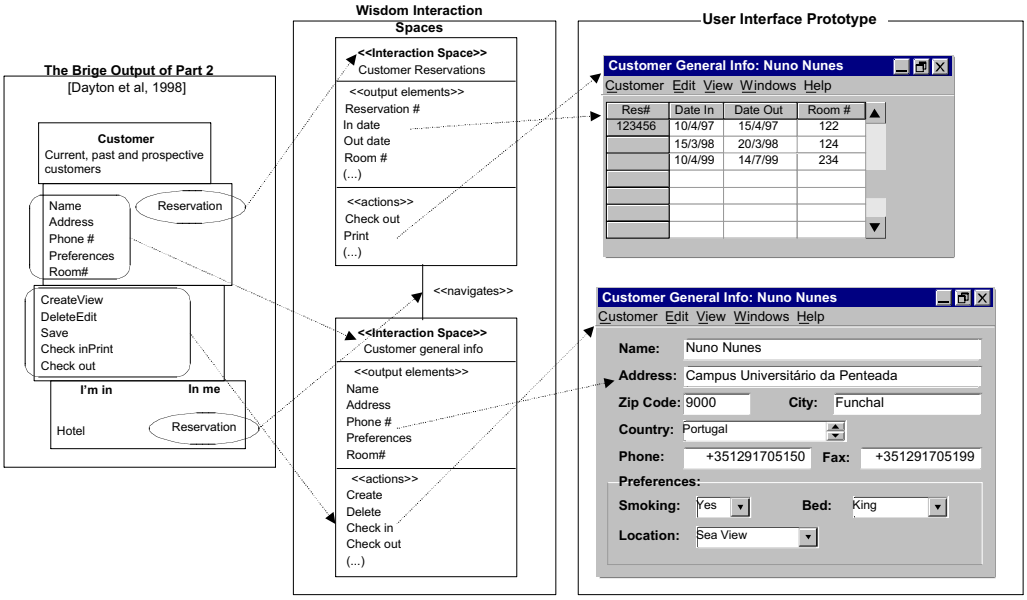


Figure IV.27 - Using the Bridge to create concrete GUIs in Wisdom

Figure IV.27 illustrates the process of mapping descriptions from the Bridge participatory sessions into the Wisdom model based approach and the concrete user interface. Leftmost in the figure is a Bridge task object with corresponding index cards showing the object identity, attributes, actions and containment relationships. To the left is the Wisdom formalization of that information into stereotyped interaction spaces. The different interaction spaces of the Bridge task object map each one to an individual interaction space in Wisdom. Then, attributes are mapped into stereotyped input or output elements, actions to stereotyped actions, and containment relationships to navigate or containment stereotyped associations. This mapping clearly shows the role of Wisdom interaction spaces realizing the interface architecture. The final step in this process would be to execute the Bridge part 3 to map the Wisdom interaction spaces into concrete GUI objects. This mapping is show rightmost in the illustration. Refer to [Dayton et al., 1998] for more details on the Bridge part 3.

IV.6. DIFFERENCES BETWEEN WISDOM AND OTHER UC-OO METHODS

Since all the methods described in section III.4.2 are UC-OO methods, they all comply to some extent with the UC-OO framework and underlying useful models that support user-centered development and user-interface design as described in section III.3. The main differences between Ovid, Idiom, Usage-centered Design (see sections III.4.2.1, III.4.2.2 and III.4.2.3) and Wisdom are related to the modeling support that each method provides for each useful model in the UC-OO framework. In other words, the different methods stipulate a specific set of modeling constructs to convey the information on the useful models for user-centered development and user-interface design.

Models / Artifacts	Ovid	Idiom	Usage-centered Design	Wisdom
User profile	-	User categorization *	User role model	User role model
Model of existing tasks	Task analysis output *	Course-grained task model *	Essential use-case model	Use case model
Domain model	User's Model Designer's Model	Domain model Core model	Domain model	Domain model
Business process model	-	-	Operational model	Business model
User interface architecture description	-	-	-	Interaction model
Model of envisioned tasks	Task descriptions	Fine grained task model	Essential use-case narratives	Essential task flows
Model of user-interface presentation	View model	View object model	Content model	Presentation model
Model of user-interface behavior	View state model	View dynamic model	-	Dialogue model
Model of concrete user-interface	Implementation *	Concrete user-interface design *	Visual and interaction design *	Evolving Prototypes *

*** - Denotes that the method is not prescriptive regarding the model.**

Figure IV.28 – Comparison between different UC-OO methods and Wisdom with respect to the useful model for user-centered development and user-interface design.

Figure IV.28 illustrates how the different UC-OO models compare to Wisdom with respect to the different useful models for user-centered development and user-

interface design (see section III.3). This table is a summary of the different models for each method as reviewed in sections III.4.2.1, III.4.2.2 and III.4.2.3.

As the table in Figure IV.28 illustrates, Ovid is the only method lacking an explicit reference to user profiling. Idiom references user categorization (a simplified form of profiling) but doesn't provide any specific modeling constructs or techniques to support the activity. Usage-centered Design provides modeling support for user profiling in the form of user role maps that convey the different user roles and their relationships. Furthermore, User-centered Design provides forms for specifying different profiling information (incumbents, proficiency, interaction, information, usability criteria, functional support and other criteria as described in section IV.4.1 and [Constantine and Lockwood, 1999]). Wisdom adapts the original proposal of Usage-centered Design in way compatible with the current UML standard. The resulting user role model in Wisdom is a set of UML extensions that, not only distinguish between human and system actor, but also provide further stereotyping of class attributes for different profile information.

As we mentioned in section III.3.1, both Ovid and Idiom are not prescriptive regarding task analysis, that is, models of existing tasks. Usage-centered Design provides essential use cases and essential use-case narratives (see sections III.4.2.3 and IV.2.1.2). Wisdom follows the same principle but promotes a diagrammatic representation of essential task flows with UML activity diagrams. The differences between the two approaches are described in detail in section IV.2.1.2.

All the methods, including Wisdom, prescribe the use of object models to represent the information in the domain model. Idiom is the only method providing a distinction between the domain model and the core conceptual model. As mentioned previously that distinction can be supported in Wisdom through the use of different UML views, that is, using different projections of the domain model (see section III.3.3).

Ovid and Idiom don't mention a business process model as an important model for user-interface development. Usage-centered design defines the operational model, although with a different meaning from what is defined in the UC-OO framework in section III.3.2. The operational model in Usage-centered Design involves the salient factors in the operational context that affect the user-interface (notably environment profiles). Those factors are related to the conditions in which the envisioned system will operate and don't include specifically the business processes of the underlying organization. Furthermore Usage-centered design captures that information in natural language. Therefore, Wisdom is the only UC-OO method explicitly endorsing a business process model as defined in section III.3.2.

Wisdom is the only method providing explicit modeling constructs to convey user-interface architecture. There are several references in Usage-centered Design to user-

interface architecture, but they only refer to design level constructs, that is, models of presentation and navigation aspects. The Wisdom user-interface architecture (described in section IV.3.2), not only leverages well-grounded work on interactive system architectures (see section II.6), but also provides an approach that seamlessly expands and integrates the existing OO analysis architectural pattern (see section II.4.3.2).

At the design level all UC-OO methods provide modeling support for the presentation aspects of interactive systems. The differences, discussed in section IV.4.5, are mainly related to the way the modeling constructs are classified and structured. Ovid emphasizes the containment relationships between (object) views and the relationship to user perceivable objects (domain objects). Idiom roughly follows the same approach of Ovid, but provides additional support for structural and behavior descriptions of views. Usage-centered Design focuses more on the navigational aspects (navigation maps). Ovid, Idiom and Usage-centered Design have a classification schema closely related to the WIMP interaction style. Wisdom differs from all the other methods providing a more abstract notion of the presentation-modeling construct (interaction space). Furthermore, Wisdom provides stereotyped input and output elements, and also actions that enable the specification of additional semantics useful for automatic user interface generation.

Regarding models of user interface behavior, both Ovid and Idiom focus on the internal behavior of view (the view behavior models) and promote the use of UML behavior diagrams (sequence diagrams) do convey the inter-view behavior. This approach attaches the user-interface behavior, and to some extent the modeling of the envisioned tasks, to the presentation components of the user-interface. Furthermore, UML behavior diagrams are recognized to have problems conveying usable descriptions of moderate size envisioned tasks (see section III.4.3.2). Usage-centered design lacks support for dynamic aspects of the user interface. Wisdom is the only UC-OO method providing support for a specific notation for task modeling to convey the behavior of the user interface. Not only Wisdom supports one of the most qualified task notations in the usability engineering field (ConcurTaskTrees), but also promotes the integration with the UML, thus enabling the relationship of task models with other modeling constructs.

All the existing UC-OO methods, including Wisdom, don't provide modeling constructs for concrete user-interfaces, that is, models capable of conveying all the information required to automatically generate the actual user-interface. UC-OO approaches are traditionally aimed at analysis and design level models, and rely on user-interface development toolkits to support interaction design. Wisdom follows the same approach with respect to implementation-oriented models, however several experiences (outlined in the next chapter) demonstrate the potential of generating concrete user interfaces from highly conceptual models such as the ones provided by the presentation and task models.

IV.7. CONCLUSION

This chapter presented the Wisdom method, including the application context of small software developing companies and the major original contributions of the thesis: the Wisdom process, architectural models and notational extensions. The chapter ended with a discussion of the differences between Wisdom and other UC-OO methods (Ovid, Idiom and Usage-centered Design).

Wisdom is a lightweight software engineering method specifically tailored for developing interactive applications by small software development groups required to take advantage of enhanced communication, flexibility, fast reaction, improvisation and creativity to sustain the highly turbulent and competitive environments in which they operate.

The Wisdom process is a new OO-UC software development process framework that recognizes the needs of software engineering in the small and takes advantage of what best characterizes this development context. The Wisdom process is based on an evolutionary prototyping model that best deals with the changing requirements and critical time to market factors that SSDs usually face. Evolutionary prototyping is also a model that seamlessly rationalizes the chaotic ways of working characterizing many SSDs, thus enabling process improvement strategies. The Wisdom process provides specific techniques to prevent the well-known problems of evolutionary approaches. In particular Wisdom includes a set of UML-based architectural models that prevent evolving software intensive systems from becoming badly structured. In addition, Wisdom includes an essential use-case and task flow driven approach that focuses development on the real-world tasks. Essential use-cases and task flows play a central role in the Wisdom process. Task flows emerge from participatory sessions and guide the development process ensuring that evolving prototypes are prioritized in terms of the real-world tasks that provided the critical added value for the customers and end-users. Therefore, Wisdom differs from the conventional SE approach that focus on internal functionality and typically leads to complex systems that are hard to use.

Wisdom also provides two architectural models that leverage interactive system development. The Wisdom model architecture specifies the different models of interest to effectively develop interactive systems. This model architecture guides the thought process in Wisdom, from the high level models built to capture requirements, to the implementation models that fully specify the final system. In addition, the Wisdom model architecture enables the separation of concerns between the internal functionality and the user-interface, both in terms of presentation and behavior aspects. At the analysis level, Wisdom promotes a new user-interface architecture that

expands the understanding of the existing UP analysis framework to include usability-engineering concerns. The Wisdom user-interface architecture enables developers to specify robust and reusable architectural models of interactive systems.

The Wisdom notation supports the Wisdom architectural models and the different modeling techniques used to detail those models. The Wisdom notation involves both a subset of the different UML modeling constructs, and a set of extensions in the form of a UML profile. The Wisdom notation reduces the number of UML diagrams required to document interactive systems, thus it only required roughly 29% of the total number of UML concepts. The main contributions at the notational level are concentrated on the presentation and dialogue models. At the presentation level Wisdom enables modeling constructs to specify conceptual user-interfaces around a central notion of an interaction space. Interaction spaces convey the structure of the user interface, and can be combined with containment and navigational relationships, thus providing and accurate model of the presentation aspects of the user-interface. At the dialogue level Wisdom enables the specification of usable and complex task structures based on a UML extensions that accommodates one of the most widely used task notation in the usability-engineering field. Moreover, Wisdom also provides UML extensions to support user role modeling, an important activity in user-centered development. All the notation extensions in Wisdom comply with the UML standard. Therefore Wisdom is the only UC-OO method fully compatible with the UML and taking advantage of the associated tool support.

The next chapter describes several experiences of applying the Wisdom method in different aspects related to tool issues and UI patterns.

V. PRACTICAL EXPERIENCE AND RESULTS

"Engineering is the art or science of making practical."
—Samuel C. Florman (1976)

This chapter describes practical experience of applying the Wisdom method in different aspects related to user interface design and specifically the impact regarding tool support.

Section 1 discusses tool issues. The section starts with a brief description of CASE tools and their role in software development. In particular we discuss the classification of CASE technology, including several studies about CASE usage, and how the UML can influence and benefit the CASE tool market. We then briefly present the UML interchange format that brings a new perspective into tool interconnection mechanisms. We discuss the importance of the OMG's XML Metadata Interchange (XMI) standard to leverage tool interoperability and briefly outline the XMI architecture, usage scenarios and document structure. The second part of the section focuses on user interface tools and presents a classification of such tools, including an historical perspective and future trends in the field. This subsection stresses the new requirements for user interface tools that emerge from the appearance of multiple information appliances that will ultimately influence the requirements, complexity and diversity of future user-interfaces. This theme is discussed in detail in the following section about appliance-independent user interface description languages.

Section 2 discusses several experiences automatically generating appliance-independent user interfaces from Wisdom models. The section starts describing the

importance of automatic generation technology with the advent of multiple information appliances. We then describe one of the first languages that support automatic rendering of user-interfaces for multiple devices. The Abstract User-Interface Markup Language (AUIML), developed at the IBM Ease of Use group was used in several experiences we conducted to evaluate the potential of using the XMI format to automatically generate concrete user interfaces from the Wisdom presentation model. We then discuss how those experiences could be generalized to the forthcoming XForms W3C standard that defines the next generation web forms. Specifically we present a proposal for a multiple tool environment that supports flexible automatic generation of user interfaces for different information appliances.

Section 3 discusses how tool support for task modeling can be enhanced through the benefits of the UML interchange mechanisms. This section illustrates how the XMI interchange format can foster tool interoperability enabling two different notations to be interchanged between UML modeling tools and task modeling tools. We illustrate how we can enhance the support for editing, simulating and perform completeness checking on CTT models that can be transferred without loss of information back and forth between UML and the CTT environment (CTTe). This section ends with a new proposal for an environment that integrates, the experiences with appliance independent automatic generation of user-interfaces, with the tool support for task modeling (CTTe). We explain why task models can complement presentation based automatic generation in such an environment.

Section 4 discusses the use of the Wisdom notation to express user interface patterns. This section discussed the increasing importance of user-interface patterns to deal with the growing complexity of modern user interface design. We argue that one of the major problems preventing the dissemination of user-interface patterns is the lack of a notation that enables the representation of abstract solutions underlying the patterns. We then illustrate how the Wisdom notation, for both the dialogue and presentations models, can be used to fill this gap increasing the expressive power of the recurring solutions conveyed in user-interface patterns. This section ends with a discussion about the importance of user-interface patterns as design aids for user interface designers. We argue that using standard notations, like the ones proposed in Wisdom, can enhance tool support for user-interface design. Particularly enabling tools to provide rapid and flexible access to user-interface patterns when they can be most effective, that is, when designers are modeling the application.

V.1. TOOL ISSUES

Computer-Aided Software Engineering (CASE) tools are expected to increase productivity, improve quality, easier maintenance, and make software development more enjoyable [Jarzabek and Huang, 1998]. However, several published results suggest that CASE tools are seldom adopted in software organizations, and fail to deliver the benefits they promise [Kemerer, 1992; IIvari, 1996; Jarzabek and Huang, 1998; Sharma and Rai, 2000]. Kemerer, for example, reports that one-year after introduction, 70% of the CASE tools are never used, 25% are used only by a limited number of people in the organization, and 5% are widely used by only one group of people but not to capacity [Kemerer, 1992]. Despite of limited CASE tool adoption, there is evidence that the technology improves, to a reasonable degree, the quality of documentation, the consistency of developed products, standardization, adaptability, and overall system quality [IIvari, 1996].

One of the major goals of the UML was to encourage the growth of the object tool market, by providing a common set of concepts, which can be interchanged between tools and users without loss of information. According to [Robbins, 1999], that goal is very close to a reality. The International Data Corporation (IDC), a market research firm that collects data on all aspects of the computer hardware and software industries, has published a series of reports on object-oriented analysis and design tools: "IDC expects revenues in the worldwide market for OOAD tools to expand at a compound annual growth rate of 54.6% from $127.4 million in 1995 to $1,125.2 million in the year 2000" [Robbins, 1999]. Furthermore, IDC expects that much of this growth to stem from adoption of OOAD tools by smaller software development organizations.

CASE technology can be broadly classified in terms of their support to development tasks into three major areas, as follows [Sommerville, 1996]:

- Support for development activities such as requirements classification, analysis, design, implementation and test. This kind of CASE technology is very mature for low-end activities – for instance compilers, editors, debuggers and integrated development environments (IDEs). Tool support for high-end activities usually involves a close relationship with specific development methods, languages and technologies - for instance, object-orientation, entity-relationship modeling, CORBA, UML and the Unified Process. Therefore, the success of high-end CASE technology is highly influenced by the lack of standards for modeling languages, associated methods and interchange formats – one of the specific problems the UML and related standardization initiatives aim to solve;

- Support for process management activities such as process modeling, metrics, risk management, process planning and scheduling, configuration and change management. This kind of CASE technology is to some extent mature in specific domains, for instance, version and configuration management; but is highly dependent on CASE support for specific development activities in order to leverage support for management activities;
- Meta-CASE is a technology aiming at generating specific CASE tools to support development activities or process management - usually enabling users to extend or customize their own tools from a common meta-facility. Meta-CASE is still mainly a research technology and the existing products are either complex or not widely available. Meta-CASE also highly depends on standards for meta-facilities, such as the MOF (see section II.3.2), that provide the set of basic concepts from which specific standards can be built, leveraging semantic interoperability and interchange formats that the resulting CASE technology can take advantage of.

Despite this useful classification, the literature on CASE tools conventionally restricts the term to high-end tools to support development activities and tools to support process management activities. To avoid misconceptions here we use the term CASE in that conventional way.

Several studies have been carried out to assess CASE tool adoption and infusion in software development organizations [IIvari, 1996; Sharma and Rai, 2000]. CASE adoption refers to the proportion of development tasks for which CASE tools are used at or beyond the experimental level [Sharma and Rai, 2000]. CASE infusion refers to the extent to which CASE is used to support development tasks [Sharma and Rai, 2000]. In a survey of 105 Finnish software development organizations, including both public and private organizations of various sizes and market segments, Iivari studied the impact of different factors in CASE usage and effectiveness [IIvari, 1996]. This study pointed out that voluntariness (the degree to which an innovation is perceived as being voluntary) is the most influencing factor of all (in this case with a negative contribution). Management support and relative advantage (the degree to which an innovation is perceived as better than its precursor) had also a significant positive influence in CASE adoption. The author relates the high negative impact of voluntariness, revealed by the study, with the importance of individualism over management influence, namely in countries or organizations where traditionally the power distance is low [IIvari, 1996]. The author conjecture about the impact of individualism is consistent with the fact that, in the same study, 85% of the projects averaged 1 to 19 persons – an evidence also consistent with our profile of small companies and development groups within user organizations (see section IV.1). In terms of the impact of CASE tools in productivity and quality of software development, the survey points out that CASE usage emerges as a significant predictor of both perceived productivity effects and quality effects [IIvari, 1996]. In general the results of this study point out that high CASE usage tends to increase the

productivity and quality of software development, and that high voluntariness tends to decrease CASE usage.

Another study of CASE deployment in US software development organizations, assessed tool usage in terms of support for production, coordination and organization – a framework that roughly follows the classification provided in the beginning of this section. Sharma and colleagues surveyed 350 public and private organizations, also of various sizes and market segments, and concluded that the lowest percentage of CASE adoption is in software departments with 1-10 employees. The highest percentage of CASE adoption is in the segment of 11-50 employees, and the percentage of adopters doubles that of non-adopters in larger organizations [Sharma and Rai, 2000]. CASE adoption and infusion levels for adopter organizations averages 0.6/1.0 for adoption and 0.5/1.0 for infusion, meaning that adopter organizations are using CASE for 60% of development tasks and that CASE usage is limited to a small segment of projects (0.5/1.0). Adoption and infusion levels are balanced in terms of CASE support for production (0.68 and 0.54 respectively) and organization (0.65 and 0.48 respectively) but fairly low for coordination (0.45 and 0.48 respectively). CASE adoption levels for production tasks are very unbalanced, ranging from 0.39 for testing and validation tasks, to 0.72 for design and construction tasks, and 0.95 for representation and analysis tasks. In general the results of this study point out that CASE is mainly used in large organizations (>50 employees), for only slightly more than half of the development tasks (adoption level), and only in a very limited subset of projects (infusion level) [Sharma and Rai, 2000].

The studies mentioned before, clearly characterize the different problems with CASE tool adoption, infusion and usage in terms of development tasks. They also point out that CASE tools increase the productivity of software developers and the quality of the process and end products. However, few indications are given with respect to why CASE tools are not widely adopted, in particular, for small companies and different development tasks. Jarzabek and Huang, based on their extensive experience designing and using CASE tools, claim that existing CASE tools are far too oriented to software modeling and construction methods, and give little attention to other factors that matter to programmers [Jarzabek and Huang, 1998]. Creative, problem-solving aspects of software development receive little support from CASE tools and therefore they are not attractive to software developers. The authors claim that CASE tools should support those (soft) aspects while enabling a seamless transition to more rigorous (hard) aspects. Moreover, the tools should provide a natural process-oriented development framework, rather than a method-oriented one, thus playing a more active role in software development [Jarzabek and Huang, 1998]. An outline of this approach, called by the authors' user-centered CASE tools, is depicted in Figure V.1. The user-centered vision of CASE tools is clearly more consistent with the requirements of small software development groups as described in section IV.1. Particularly important is the focus on creativity and improvisation as key success factors in these environments.

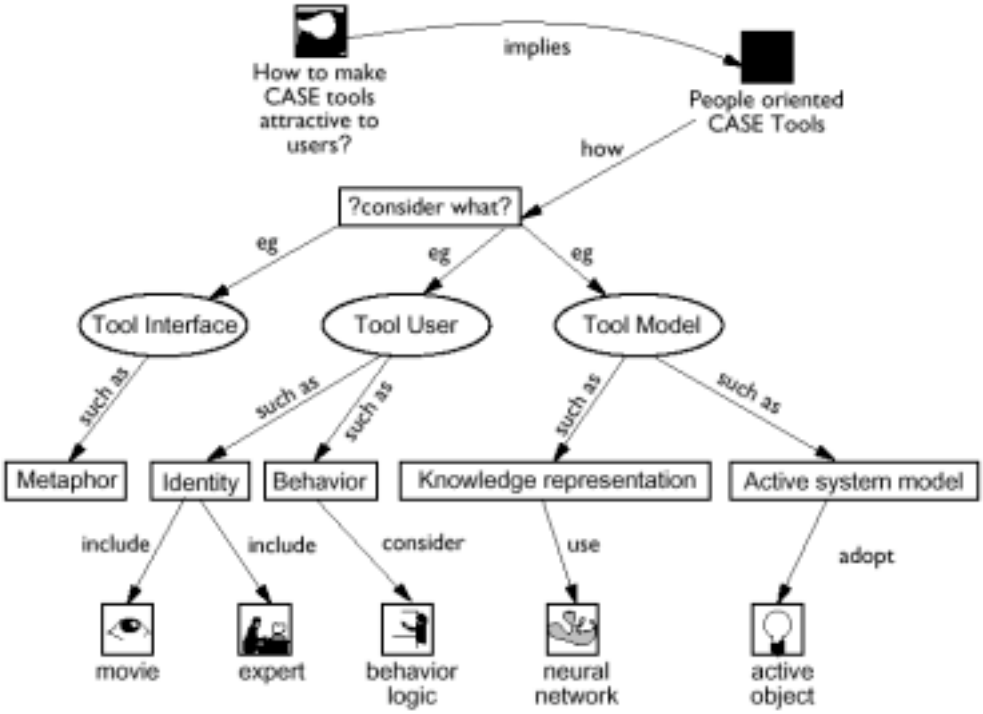


Figure V.1 – Jarzabek and Huang perspective on how to make CASE tools attractive to developers (source: [Jarzabek and Huang, 1998])

In the following sections we describe key technologies and approaches that could leverage this new vision of CASE tool support. In particular we focus on the UML interchange support to enhance tool interoperability, thus, enabling different highly focused tools to support developers in the different tasks they must accomplish in order to develop highly usable interactive software.

V.1.1.The XMI Interchange Format

The XML Metadata Interchange (XMI) standard defines an alternate physical specification of the UML in the eXtensible Markup Language (XML) [W3C, 1999b], specifically for interchanging UML models between tools and/or repositories. The main purpose of XMI is to enable easy interchange of metadata between modeling tools (based on the OMG-UML) and metadata repositories (OMG-MOF based) in distributed heterogeneous environments [OMG, 2000]. XMI integrates three key industry standards:

- XML - eXtensible Markup Language, a W3C standard;
- UML - Unified Modeling Language, an OMG modeling standard;
- MOF - Meta Object Facility, an OMG metamodeling and metadata repository standard.

The XMI standard enables different types of application development tools to interchange their information. Examples of such application include [Brodsky, 1999]:

- Modeling tools – such as object-oriented UML tools, for instance Rational Rose and TogetherJ [Rational, 2000; TogetherSoft, 2000];
- Development tools – such as integrated development environments, for instance VisualAge for Java and Symantec Café [IBM, 2000a; Symantec, 2000];
- Databases, Data Warehouses and Business Intelligence tools (data mining, OLAP, etc.);
- Software assets – such as program source code (C, C++, and Java) and CASE tools, for instance WindRiver SniFF+ [Windriver, 2000];
- Repositories – such as IBM VisualAge TeamConnection and Unisys Universal Repository [IBM, 2000b; Unisys, 2000];
- Reports, report generation tools, documentation tools, and web browsers.

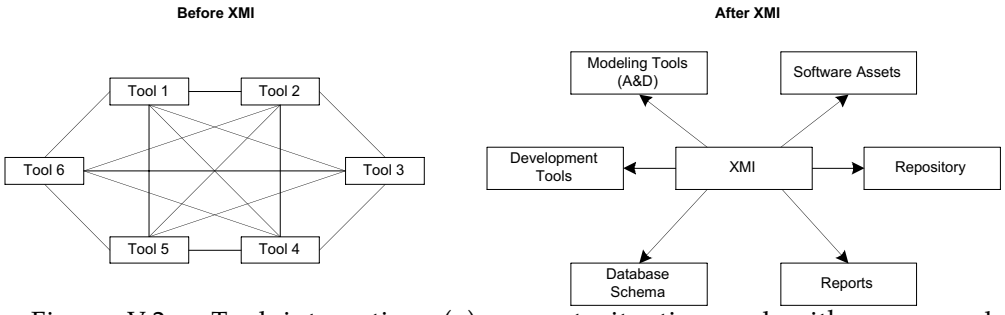


Figure V.2 – Tool integration: (a) current situation and with open model Interchange (XMI) [Brodsky, 1999]

Figure V.2 illustrates the improvement of an open and standard tool integration facility. Without an open interchange standard such as XMI, each tool was required to know the formats used by any other tool in the market. Therefore, tool vendors were obliged to develop an exceeding number of N*N-N bridges between N tools required to interchange information (left hand side in Figure V.2). The problem increases with different versions of each tool and different release and schedules for proprietary formats. With the advent of XMI, not only tool interoperability becomes simpler, but also more cost-effective (right hand side in Figure V.2). Therefore, tool vendors, integrators and end-users benefit from reduced complexity, reflected in lower costs and time to market [Brodsky, 1999].

An example of how XMI can be used follows the architecture at the right hand side in Figure V.2. A business analyst makes a UML business model using an OO modeling tool. The design is expressed in XMI and used by a software developer working on a specific programming language with an integrated development environment (IDE). Reports and documentation can be published to the web, generated directly from the XMI specification. By accessing the design in XMI, database schemas and data warehouses may be created by database designers and business intelligence analysts. This example illustrates how XMI enables users to focus directly on their roles, working as a team in an open, distributed environment. Users can employ the right products for each role, and interchange their designs in XMI using the Internet and

the transformation facilities available for the XML, that is, the eXtensible Stylesheet Language (XSL) [Brodsky, 1999].

V.1.1.1.XMI Architecture

XMI, together with MOF and UML form the core of the OMG metadata repository architecture. This architecture, illustrated in Figure V.3, includes the following key aspects [OMG, 2000]:

- A four-layered metamodeling architecture for general-purpose manipulation of metadata in distributed object repositories (see section II.3.2);
- The use of MOF to define and manipulate metamodels programmatically using fine-grained Common Object Request Broker Architecture (CORBA) interfaces. This approach leverages the strength of CORBA distributed object infrastructure [OMG, 2001];
- The use of UML notation for representing models and metamodels (see section II.3.2);
- The use of standard information models (UML) to describe the semantics of object analysis and design models;
- The use of Stream-based Model Interchange Format (SMIF) - the current XMI proposal - for stream based interchange of metadata.

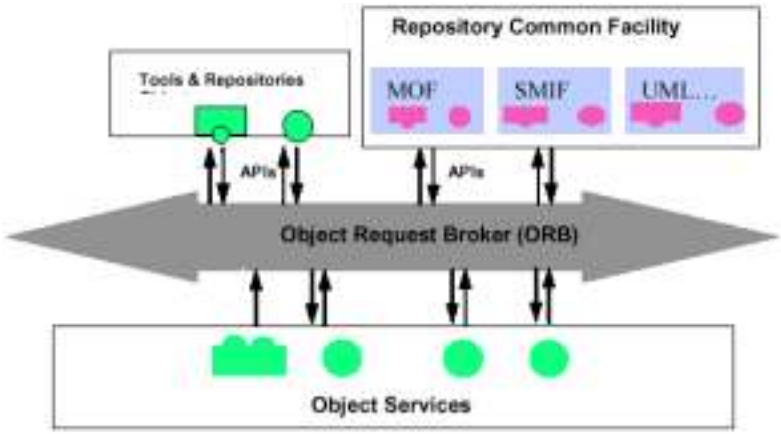


Figure V.3 – The OMG Repository Architecture and the SMIF (XMI) Interchange Standard (source: [OMG, 2000])

V.1.1.2.XMI Usage Scenarios

The main usage scenarios for the XMI interchange format are as follows [OMG, 2000]:

- Combining Tools in a Heterogeneous Environment - since no single tool exists for both modeling the enterprise and documenting the applications that implement the business solution, a combination of tools from different vendors is necessary. However, this combination leads to complex translations or manual re-entry of information. XMI eases the problem of tool interoperability by providing a flexible and easily parsed information interchange format. Since the XMI specification

contains the definitions of the information being transferred, as well as the information itself, a tool reading the stream can better interpret the information content. Furthermore, including the definitions in the stream enhances the flexibility of the scope of the information - it can be extended with new definitions as more tools are integrated to exchange information;

- Cooperating with Common Metamodel Definitions - if the tools exchanging information share the same meta-model, all the information transferred can be understood and used without limitations. Furthermore, since the MOF meta-model is expressed using itself (see section II.3.2), XMI can also be used to share meta-model definitions, thus promoting the use of common meta-models by different tool providers. In addition, each vendor must be able to extend the information content of the model to include items of information that have not been included in the shared model. XMI allows a vendor to attach additional information to shared definitions in a way that allows the information to be preserved and passed through a tool that does not understand the information. Loss-less transfer of information through tools is necessary to prevent errors that may be introduced by the filtering effect of a tool passing on only that information it can understand itself. Using this extension mechanism, XMI stream can be passed from tool to tool without suffering information loss;
- Working in a Distributed and Intermittently Connected Environment - the use of XMI for a metadata interchange facilitates the exchange of model and design data over the Internet and by phone, even with restricted connectivity to the network and limited bandwidth. Appearing as a set of hyper-linked Internet documents, the data can be transferred and transported easily. The same mechanism can be used to upload modifications. The type definitions can be separated from the actual content and versioned to allow consistency checking;
- Promoting Design Patterns and Reuse - It is often difficult to develop and exploit best practices across the development group and organization without being able to exchange model and design data between different tool sets. XMI only requires the development team to agree on the meta-models for the data to be shared, plus a standard mechanism for extending that meta-model with their own types of meta-data. Therefore, it does not require the tool vendors to invest in the same technology stack. Vendor extensions to a standard meta-model are designed to enable other vendor's tools to process and use the standardized information while being able easily retain and pass through vendor specific extensions.

V.1.1.3.Overview of XMI DTD and Document Structure

An XML Document Type Definition (DTD) provides a means by which an XML processor can validate the syntax and some of the semantics of an XML document. Although the use of DTDs is optional, it improves the confidence in the quality of the document [OMG, 2000].

Every XMI DTD contains the elements generated from an information model, plus a fixed set of element declarations that may be used by all XMI documents. These fixed elements provide a default set of data types and the document structure, starting with the top-level XMI element. Each XMI document contains one or more elements, called XMI, that serve as top-level containers for the information to be transferred. XMI is a standard XML element and may stand alone in its own document or may be embedded in XML or HTML documents.

Every XMI DTD consists of the following declarations [OMG, 2000]:

- An XML version processing instruction: `<? XML version="1.0" ?>;`
- An optional encoding declaration that specifies the character set, which follows the Unicode standard: `<? XML version="1.0" ENCODING="UCS-2" ?>;`
- Any other valid XML processing instructions;
- The required XMI declarations;
- Declarations for a specific meta-model;
- Declarations for differences;
- Declarations for extensions.

Every XMI document consists of the following declarations [OMG, 2000]:

- An XML version processing instruction;
- An optional encoding declaration that specifies the character set;
- Any other valid XML processing instructions;
- An optional external DTD declaration with an optional internal DTD declaration: `<! DOCTYPE XMI SYSTEM "http://www.xmi.org/xmi.dtd">;`

The top level XML element for each XMI document is the XMI element. Its declaration is as follows:

```
<!ELEMENT XMI (XMI.header?,
               XMI.content?,
               XMI.difference*,
               XMI.extensions*) >
<!ATTLIST XMI xmi.version CDATA #FIXED "1.1"
               timestamp CDATA #IMPLIED
               verified (true | false) #IMPLIED >
```

The XMI element contains the following structural elements:

- `XMI.header` - contains version declarations and optional documentation regarding the transfer. The `XMI.header` element is optional in XMI 1.1, and contains XML elements, which identify the model meta-model, and meta-meta-model for the metadata; as well as an optional XML element that contains various information about the metadata being transferred;
- `XMI.content` - contains the core information that is to be transferred. The `XMI.content` XML element contains the actual metadata being transferred. It

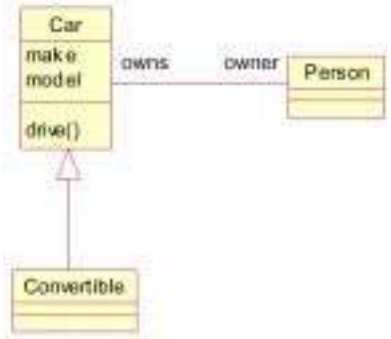
may represent model information or meta-model information. Its declaration is `<!ELEMENT XMI.content ANY>;`

- Differences - specify the differences between two XMI documents to be transferred. This is useful in cases such as small changes to a large set of information;
- Extensions - allow the transfer of private tool information beyond that already present in a DTD. The extensions are especially useful for round-trip exchanges between tools that benefit from preservation of private additional information due to their own internal architecture;
- The `xmi.version` - defines the XMI version. The `timestamp` indicates the date and time that the metadata was written. The `verified` attribute indicates whether the metadata has been verified ("true," means the verification of the model was performed by the document creator at the full semantic level of the meta-model).

Generating element declarations provides an architectural consistency for each element in several important areas: Object identity, extensibility, navigation, and linking. This consistency enables tools to traverse any XMI document or DTD in a regular manner to find the information needed [Brodsky, 1999]:

- Object identity is standardized through the `xmi.element.att` macro (called an XML entity), which declares markers for optionally declaring unique identity via universal identifiers (`uuids`), plus optional shorthand labels and local document identifiers (`xmi.id`);
- Extensibility is standardized by the optional `XMI.extension` element which allows nested extensions in addition to those present in the extensions section;
- Navigation is standardized by the regular pattern from which DTDs are generated;
- Linking is standardized through the `xmi.link.att` macro, compatible with the upcoming XLink and XPointer specifications from the W3C [W3C, 2000c] to provide both internal and external linking by identifiers (ids) and references (hrefs). In addition, every element may contain a reference to another location for its definition, allowing transparent and consistent linking.

Figure V.4 illustrates a minimal example of XMI for a UML car class. The resulting document contains two sections (header and contents) following the structure described before. Relationships between model elements are described using the XMI linking mechanisms, for instance as required to express the associations between class elements in the example provided in Figure V.4.




```
<XMI xmi. version="1.1"
xmlns:uml="org.omg/uml1.3">
<XMI. header>
<XMI. documentation>
An example of a car
</ XMI. documentation>
<XMI.metamodel name="UML" version="1.3"
href=" uml1.3.xmi"/>
<XMI.model name="Cars" version="1.0"/>
</XMI.header>
<XMI.content>
<Class name="Car">
<Class.ownedElements>
<Attribute name="make"/>
<Attribute name="model"/>
<Operation name="drive"/>
</Class.ownedElements>
</Class>
</XMI.content>
</XMI>
```

Figure V.4 – XMI Example for a Car class (adapted from [Brodsky, 1999])

XML validation can determine whether the XML elements required by this specification are present in the XML document containing meta-model data. XML validation can also perform some verification that the meta-model data conforms to a meta-model. However, it is impossible to rely solely on XML validation to verify that the information transferred satisfies all of a meta-model's semantic constraints.

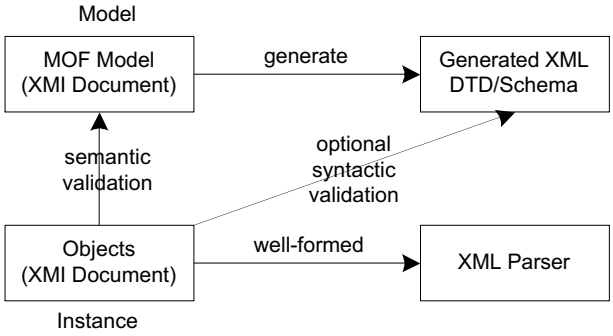


Figure V.5 – Validation of XMI documents (adapted from [Brodsky, 1999])

XMI is not restricted to express UML information. The OMG is working on a standard, called Common Warehouse Metadata (CWM), to enable the expression of database information and schemas in a common form that can be interchanged with the XMI. The OMG's CWM interchange initiative will provide a general definition for the interchange of relational and hierarchical databases, database management and data warehousing. Other examples of information that can be interchanged with XMI include: Java and C++ class definitions, enterprise Java beans and electronic commerce standards. Ultimately all the meta-models that are MOF compliant (see section II.3.2) can be expressed and interchanged with XMI.

V.1.2.User Interface Tools

Virtually all applications today are built using some form of user interface (UI) tool [Myers et al., 2000]. Contributions from research on UI tools and technology had a tremendous impact on the current practice of software development – for instance

object-oriented programming, event languages and component-based development were all contributions related to UI research. Moreover, UI tools are an important segment of the tool market, accounting for 100 million USD per-year [Myers et al., 2000].

A user interface tool is any software aimed to help create the part of a software intensive system that handles the input and output between the user and the interactive system (the user interface). User interface tools have been called various names over the years. The most popular terms are User Interface Management Systems (UIMS), Toolkits, User Interface Development Environments (UIDEs), Interface Builders, Interface Development Tools and Application Frameworks [Myers, 1995; Myers, 1996; Myers et al., 2000].

User interface tools bring many advantages to interactive system development. According to Myers [Myers, 1995] the main advantages can be classified as follows:

- UI tools increase the quality of the user interfaces – mainly because tools enable rapid prototyping of UIs and subsequent incorporation of changes discovered through user testing. They also leverage multi-UI creation, foster consistency and enable different specialists (other than programmers) to be involved in the UI design process;
- UI tools promote UI code that is easier and more economical to create and maintain – because UI specifications can be represented, validated, and evaluated more easily. In addition, UI tools reduce the code to write, better modularization, separation of concerns (between internal functionality and UI), reduced complexity, increased reliability, and increased portability between platforms.

V.1.2.1.Classification of UI Tools

A user interface software tool may be divided into various layers, a typical approach is described by Myers and involves: the windowing system, the toolkit, and higher-level tools (see Figure V.6) [Myers, 1995; Myers, 1996]. The windowing system supports the separation of the screen into different regions (conventionally called windows). Windowing systems are usually an integral part of the operating system (a notable exception is XWindows) and therefore examples of such systems can be found in many common operating systems in the use today, for instance MacOS and Microsoft Windows. The toolkit is a library of widgets (ways of using a physical input device to input a certain type of value) that can be called by the application program. Typical widgets included in a toolkit are menus, buttons, scroll bars, text fields, and so on. Toolkits have the advantage of fostering consistency between UIs developed using the same toolkit. Furthermore, toolkits provide reuse at the functional level, for instance the functions that control the basics of widgets. However, toolkits have several problems: they limit the interaction styles to those provided, and they are expensive to create and complex to manage [Myers, 1995]. Examples of toolkits are

collections of procedures (APIs) that can be called by applications (for instance SunView Toolkit, Macintosh Toolbox, etc.) and object-oriented frameworks (such as those provided by Smalltalk, Xt, Garnet, Microsoft Foundation Classes, Java Abstract Window Toolkit and Java Swing).

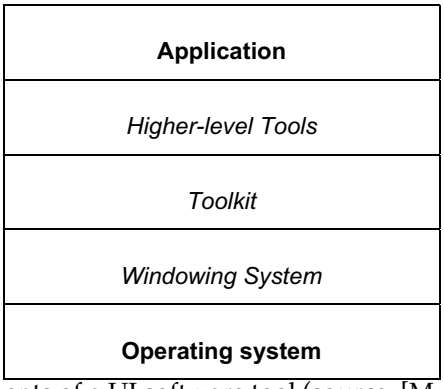


Figure V.6 – Myers components of a UI software tool (source: [Myers, 1995]

Since programming at the toolkit level is complex and difficult, higher-level tools are required to simplify and ease the process of producing the user interface [Myers, 1995]. Most high-level tools work both at design-time and runtime. The design-time component is responsible for helping design the user interface - for instance through a graphical editor that helps layout the widgets or a compiler that processes the UI specification language. The runtime component manages the UI when the user is interacting with the application and therefore is conventionally called a User-Interface Management System (UIMS). Optionally a post-runtime component can support UI metrics and evaluation by providing automatic cognitive models of the UI, for instance GOMS-based models such as the ones described in section II.1.2.2. However, to date very few tools provide this kind of post-runtime facilities [Myers, 1995].

High-level tools mainly differ in terms of the specification language and style used to specify the user interface. Myers provides a classification based on the specification format provided by high-level tools as illustrated in Figure V.7 [Myers, 1995].

Specification Format	Examples
Language Based	
State Transition Networks	VAPS
Context-Free Grammars	YAW, LEX, Syngraph
Event Languages	ALGAE, Sassafras, HyperTalk
Declarative Languages	Cousin, Open Dialog, Motif, UIL
Constraint Languages	Thinglab, C32
Screen Scrapers	Easel
Database Interfaces	Oracle
Visual Programming	LabView, Prograph, Visual Basic
Application Frameworks	
	MacApp, Unidraw, Microsoft Foundation Classes, Java AWT, Swing
Model-Based Generation	
	MIKE, UIDE, ITS, Humanoid
Interactive Graphical Specification	
Prototypes	Bricklin's Demo, Director, HyperCard
Card-based Systems	Menulay, HyperCard
Interface Builders	DialogEditor, NeXT Interface Builder, Prototyper
Data Visualization Tools	DataViews
Graphical Editors	Peridot, Lapidary, Marquise

Figure V.7 - Specifications formats for High-level User Interface Development Tools (adapted from [Myers, 1995])

Language-Based Systems

Language based systems enable designers to specify the syntax of the user interface (the valid sequences of input and output actions) using special purpose languages, such as context-free grammars, state transition diagrams, declarative or event based languages, and so on (see formats and examples in Figure V.7).

Different language-based specification formats have different advantages and disadvantages [Myers, 1995]. State transition networks, such as UML activity or statechart diagrams (see section II.3), provide a natural way of specifying the handling of a sequence of input events involved when a user interacts with a given system. However, state transition diagrams pose many problems when the UI is complex (many nodes and arcs) or when the user interacts with multiple objects at the same time, which is typically the case with modern highly interactive applications (see section II.7.2). The same problems exist with context-free grammars that are more oriented to batch processing of strings with a complex syntactic structure and thus have been abandoned for modern highly interactive user interfaces.

Event languages have several advantages over the two previous approaches, in particular the possibility of handling multiple input devices active at the same time, thus simplifying the support for non-model user interfaces. The main disadvantage is related to the complexity and distribution of the flow of control, as a result small changes can affect many different parts of the program. However, event languages are particularly efficient for small to medium sized programs.

Declarative languages take a different approach enabling the designer to focus on what should happen as opposed to how to make it happen. The main advantage of declarative languages is related to the fact that designers don't have to concentrate on the time sequence of events, but only on the flow of information. The main disadvantage is that declarative approaches are only possible for form-based UI or widget layout description, thus more complex direct manipulation techniques (such as drag and drop, drawing graphical objects, and so on) must be specified using a different approach. However, declarative languages have recently become the basis for web UI, notably HTML forms and other intent-based UI description languages discussed separately in section V.1.3.

Constraint languages allow the specification to be based on restrictions about the behavior of the UI. Constraint languages are still under research and provide a natural way to express many kinds of relationships that arise in UI, for instance that two objects must stay attached. However they require a complex runtime system to support the constraints while the user interacts with the application.

Screen scrappers are commercial tools specialized to be "front-ends" (or screen scrappers) that provide a graphical user interface for old legacy systems without changing the existing underlying functionality. Screen scrappers rely on buffering schemes and are not intended to develop modern UI from scratch.

Database interfaces are a very important class of commercial tools that support form-based or GUI based access to databases. Virtually all-major database management system (DMS) vendors provide tools that allow designers to develop UI for accessing and changing data on a database. Fourth-generation languages (4GLs) fall also into this category but are usually independent of the back-end DMS. These tools typically include interactive form editors (essentially interface builders) and special database access languages.

Visual programming provides graphical and layout as part of the program specification. Most systems that support diagrammatic languages (such as state-transition diagrams) provide mechanism for visual programming. Visual programming tools are typically easier to use by non-experienced programmers but bear the same problems of state transition schemas when the underlying UI involves many states and arcs (typical in highly interactive modeless application).

Application Frameworks

Application frameworks are the second major class of high-level tools for user interface design. Frameworks originated form the need to reduce the complexity of directly programming UI toolkits.

The first example of an application framework was MacApp [Schmucker, 1986] that provided a way to simplify the programming of the Macintosh toolbox and to comply

with the platform guidelines [Apple, 1992]. Application frameworks are typically based on a set of OO classes that can be specialized for a particular application, thus highly reducing the development effort of the common parts of the application – several studies found out that tools like the MacApp reduce the development effort in one order of magnitude [Schmucker, 1986].

Application frameworks have the drawback of still requiring designers to program the application specific subclasses, however component based approaches such as Object Linking and Embedding (OLE) [Microsoft, 2000] and JavaBeans [Sun, 2000] allow applications to be built from complex components only requiring those components to adhere to a common communication standard.

Model-based Generation

Model-based automatic generation of UIs aims at simplifying the need to specify all the details about the placement, format and design of the UI. Model-based approaches aim to generate, as much as possible, those details from higher-level models of the UI. Model-based approaches are described in more detail in section II.7. The main drawback of model-based approaches is that they failed to go beyond the research level of generating menus and dialog boxes to the commercial requirements of high-quality UIs. A further problem is that the specification languages required by those approaches are hard to learn.

Interactive Graphical Specification

Interactive graphical specification encompasses approaches that allow the definition of the UI, at least partially, by direct manipulation (placing objects on the screen using a pointing device). Interactive graphical specification is motivated by the assumption that visual presentation is of primarily importance in GUIs, thus these tools enable rapid development of rich graphical UIs by nonprogrammers. Interactive graphical specification differs from visual programming because the UI is drawn rather than generated indirectly.

Prototyping tools allow designers to quickly mock-up the visual appearance of screens. These tools don't enable full specification of the UI and thus can only be used to simulate the actual application.

Card-based system is a generic description of tools that enable the development of UIs based on sequences of static pages (also called frames or forms). Each card in these systems contains a set of widgets that are coded individually, typically through a scripting language. Unlike prototyping tools, card systems enable the specification of fully functional applications although limited in terms of the interactive facilities they provide. Interface builders (also called interface development tools) allow designers to compose dialog boxes, menus and windows from a predefined library of widgets, thus creating by direct manipulation the UI of the final interactive application.

Interface builders enable some properties of the widgets to be specified through property sheets, and even support some sequencing of windows and dialog boxes. Interface builders use the actual toolkit provided by the platform and therefore can be used to build actual parts of the application (usually generating the code). One common problem attributed to interface builders is that they provide much flexibility to developers and thus provide little guidance toward creating good UIs. Another problem is that interface builders cannot handle graphics areas or widgets that change dynamically.

Data visualization tools are an important category of tools that focus on the display of dynamically changing data. These tools are conventionally used as front-ends for simulation, process control system monitoring, network management and data analysis. Data visualization tools, unlike interface builders, usually don't use the platform widgets.

Graphical editors are tools that enable the creation of custom graphics that depend on runtime conditions, such as dynamic changing data or user actions. Graphical editors differ from data visualization tools because in the former designers draw examples of the UI that, as a result of generalization, is translated into parameterized prototypes that change in runtime.

V.1.2.2.Trends in UI Tools

In a recent survey on the Past, Present and Future of User Interface Tools, Myers and colleagues identified some issues that are important for evaluating, which approaches were successful and which ones are promising in the future [Myers et al., 2000]:

- The parts of the UI the tools address – the tools that succeeded in the past contributed significantly in one part of UI development. The majority of the successful tools and technologies focused on a particular part of the UI that was a significant problem, and which could be addressed thoroughly and effectively. Examples of successful approaches include [Myers et al., 2000]: Window managers and toolkits, event languages, interactive graphical tools, component systems, scripting languages, hypertext and object-oriented programming. Examples of approaches that failed to receive commercial success due to issues involved with trying to address the whole problem include UIMSs and Model-based and automatic generation techniques;
- Threshold and Ceiling – the threshold is related to the difficulty of learning a new system and the ceiling is related with how much can be done using the system. Successful approaches in the past are usually low threshold and low ceiling (e.g. interactive graphical tools, hypertext and the www) or high threshold and high ceiling (windows managers, toolkits and object-oriented programming). Examples of unsuccessful approaches that suffered from the high threshold problem include formal languages and constraints. Although different attempts have been made to

lower the threshold of some of those approaches, they are done at the expense of powerful features that allow for a high-ceiling, thus become less attractive for developers;

- Path of Least Resistance – tools and technologies influence the kinds of UIs that can be created, therefore successful tools lead to good UIs. Examples of successful approaches that provided a path of least resistance are window managers and toolkits. Those approaches are mainly responsible for the significant stability of the current desktop UI, which highly contributed for the consistency in today's UI and the possibility of users to build and transfer skills within different applications and platforms. Counter-examples include formal languages, which promoted rigid sequences of actions that are highly undesirable in modern non-modal UIs, thus the path of least resistance of these tools is detrimental to good UI design;
- Predictability – developers typically resist tools that can provide unpredictable results in the final systems. Most successful approaches provided predictability and control over the resulting UI, conversely the majority of tools employing automatic techniques (e.g. model based systems and constraints) made the connection between the specification and the final result difficult to understand and control;
- Moving Targets – as with any interactive system, it is very difficult to develop tools without having a significant experience and understanding of the tasks they support. Conversely, the time it takes to understand a new implementation task can lead that good tools become available when that task is less important or even obsolete. Most successful approaches took advantage of the high stability of today's standard GUI. On the contrary, nearly all-unsuccessful approaches succumbed to the moving-target problem. UIMSs, language-based approaches, constraints and model-based systems were designed to support a flexible variety of interaction styles and became less important with standardization of the desktop UI.

The discussion of the above issues for UI tools makes clear that successful approaches focused on a particular part of the user interface that was a significant problem and which could be addressed effectively reducing the development effort, allowing UIs to be created quickly and promoting a consistent look and feel. The long stability of the GUI desktop and direct-manipulation user interface style has enabled the maturing of the successful tools, alleviating the moving-target problem that affected the earliest research approaches [Myers et al., 2000]. However, the requirements for the next generation of user interfaces are quite different. The increasing diversity of computing devices and the increasing connectivity of existing desktop computers will have a profound effect on the future of UIs.

Myers and colleagues envision important implications from computer becoming a commodity, ubiquitous computing, end-user customization and the move to recognition based and 3D user interfaces [Myers et al., 2000].

Computers are becoming a commodity because the computing power availability to the general public is almost undistinguished from that available to researchers. Hence, researchers can no longer have access to high-performance computing to envision and investigate what will be available to the public in a moderate amount of time. Furthermore, the quantitative change in increased performance, made available and affordable computing in many different and embedded devices at different scales.

Computing is becoming ubiquitous [Cerf, 1992; Weiser, 1993; Abowd and Mynatt, 2000] and the move is clearly towards specialized information appliances that perform well a minimal set of tasks [Norman, 1998; Want and Borriello, 2000]. Those devices have varying input and output capabilities, for instance the conventional desktop machine varies in a factor of 4 in screen resolution (from 640x480 to 1280x1024) and in a factor of 2.5 in screen size (8 to 20 inches); conversely today's information appliances vary in a factor of 1000 in resolution and in a factor of 100 in display [Myers et al., 2000]. Moreover, information appliances are very distinct in terms of input and output devices, for example a cellular phone has a numeric keypad and voice recognition, palmtops have touch-sensitive screens and stylus and no keyboard, and so on.

Therefore, according to Myers and colleagues, the requirements of the next generation UIs entail tools that: (i) enable rapid prototyping of devices and not only software; (ii) tools that enable coordination of multiple distributed communication devices, (iii) tools that support recognitions-based interfaces (voice, gestures, etc.), (iv) tools that support three-dimensional technologies, and (v) tools that leverage end-user programming, customization and scripting [Myers et al., 2000]. In the words of Myers and colleagues "Future trends will call for new and powerful techniques (high ceiling) to deal with the anticipated dramatic increase in user interface diversity" [Myers et al., 2000].

V.1.3.Appliance Independent UI Description Languages

Although the idea of a platform independent user-interface description language is not new, the World Wide Web (WWW) provided the first effective and widely available user interface description language that is independent of the underlying platform. From the initial success of the HTML, in allowing non-programmers to design platform independent user-interfaces, there has been an explosion of different approaches for declarative device-independent UI description languages. Examples for desktop applications include Dynamic HTML (DHTML) or more specifically the interaction between HTML, Cascading Style Sheets (CSS), eXtensible Stylesheet Language (XSLT), the Document Object Model (DOM) and scripting (JavaScript, CGI, VBScript, and so on) [W3C, 1998a; W3C, 1998b]; XwingML [Bluestone, 1999] and XML-based User Interface Language (XUL) [Mozilla, 1999]. Related efforts have also emerged for different environments, notably the Wireless Markup Language (WML) [WAPF, 1999] for mobile devices with displays; and SpeechML [IBM, 1999], Voice

Markup Language (VoxML) [Motorola, 2001] and Java Speech Markup Language (JSML) [Sun, 1997] for voice-enabled devices like a conventional telephone.

This plethora of UI description languages is a consequence of the multiple hardware and software technologies involved in the development of specialized information appliances (see section V.1.2.2). This diversity in UI description languages poses the same drawbacks that UI designers faced in the early days of the personal computer. Before windowing systems, toolkits and higher-level tools (notably APIs and OO frameworks) (see section V.1.2.1) provided an abstraction layer to "hide" the device-dependent specifics. Several recent approaches have tried to solve this problem proposing a universal appliance-independent markup language to describe UIs in highly abstract way, which could be translated into more specific languages via-style sheets. Two notable examples are the Abstract User Interface Markup Language (AUIML) [IBM, 1999] and the User-Interface Markup Language (UIML) [Abrams, 1999]. A similar approach (xForms [W3C, 2001]) is being developed by World Wide Web Consortium (W3C) to try to deal with the limitations of the existing genre of "split interface" of web applications. Today's web browsers are a very basic virtual machine to render views (see MVC in section II.6) in the form of HTML documents with limited controls. The web server maintains the Model (in the MVC sense) working around the HTTP's stateless submission of form input elements by passing along the model's context by value (through hidden fields) or by reference (through cookies or HTTP login) [Khare, 2000a; Khare, 2000b]. The W3C is trying to address this problem taking advantage of the XML standard in a similar way that was already proposed in AUIML and UIML. This effort is now possible with the advent of eXtensible HyperText Markup Language (XHTML) 1.0 [W3C, 2000b], which defines the first step towards translating the basic capabilities of the HTML 4.0 [W3C, 1999a] into XML.

XHTML is a family of current and future document types and modules that reproduce, subset, and extend HTML 4. The XHTML family of document types is XML based, and thus ultimately designed to work in conjunction with XML-based user agents (a user or a system that manipulates XML documents). XHTML 1.0, the first document type in the XHTML family, provides the basis for a family of document types that will extend and subset XHTML, in order to support a wide range of new devices and applications, by defining modules and specifying a mechanism for combining these modules. This mechanism will enable the extension and sub-setting of XHTML 1.0 in a uniform way through the definition of new modules. The modularization underlying XHTML is then the strategy of the W3C to manage multiple information appliances. The process of modularization breaks the XHTML up into a series of smaller element sets that will likely be required by different platforms and recombined to meet the needs of different user communities.

In the following sections we present an overview of the AUIML description language and discuss several experiences, developed in conjunction with the IBM Ease of Use

group, to assess the possibility of generating AUIML documents from UML models. We present an overview of the markup language and briefly discuss some of the results of different experiences generating AUIML documents from both the Ovid method (see section III.4.2.1) and Wisdom models. In particular we present an approach that produces AUIML from Wisdom models through the XMI interchange format. In the last section we discuss how this approach could be generalized to the forthcoming XForms standard.

V.2. AUTOMATIC GENERATION OF APPLIANCE INDEPENDENT USER INTERFACES FROM WISDOM MODELS

As we discussed previously (see sections II.7, III.4.3.6 and V.1.2) automatic generation of user-interfaces can speed up implementation and improve productivity enabling designers to work at higher-levels of abstraction. Research on model-based UIs proved that it is possible to fully automate the process of UI design and implementation (see the different approaches discussed in section II.7). Although, commercial adoption of "pure" model-based approaches was not achieved to date (see section V.1.2.2), the technology is widespread and used by millions everyday - anytime a user opens a web-page on a browser a UI is automatically generated from a declarative HTML specification. In the words of Kovacevic "the issue is not whether automatic generation is feasible and practical, but how effective it can be" [Kovacevic, 1999]. The example of the world wide web confirms Myers' prediction that successful tools and techniques tackle only small but important issues in UI design, that is automatic generation can be effective when we limit its scope as it is the case in the WWW. It seams obvious that DHTML markup, has a successful and effective approach for describing UIs, is a low-ceiling and low threshold technology (see section V.1.2.2).

The real problem with automatic generation is, therefore, not in the technology itself but how it can be effectively used in UI design. User interface design is, to some extent, a creative activity and therefore cannot be fully automated without compromising the quality of the end user interface. Moreover, adequate user-centered design relies on continuous and iterative feedback from users and therefore automatic generation must provide support for exploration and full control over the design. It is critical that modern UI development tools enable designers to work at very high levels of abstraction, while providing support for multiple UIs but also leveraging flexibility. As Kovacevic points out, "this balance of productivity and flexibility is one issue that UI tools must resolve" [Kovacevic, 1999].

Web enabled markup languages are irrefutably the most widely used approach for describing user-interfaces for automatic generation purposes. However, hypertext markup is not adequate for modeling purposes. The complex syntax and the absence of diagrammatic representations compromise the power and flexibility of markup as an effective modeling approach. Markup languages leverage the semantic aspects in detriment of the flexibility and manipulability (by humans) of their notation. Furthermore, models of software systems are developed in a larger context that

conveys their true meaning, thus specific markup languages, such as AUIML, lack the wider context of the overall aspects of interactive system development. The next section briefly presents the AUIML appliance-independent UI description language and the following section discusses experiences generating AUIML from Wisdom models.

V.2.1.1.Abstract User-Interface Markup Language (AUIML)

The Abstract User Interface Markup Language (AUIML) [IBM, 1999] (formerly known as Druid) is an XML vocabulary developed by the IBM Ease of Use group, intended to facilitate the definition of an appliance independent UI description language – what the authors call an intent-based approach (based on the intention of the interaction as opposed to the presentation aspects). Therefore, The Druid vocabulary is intended to be independent of the client platform on which the user interface is rendered, the implementation language and the user interface implementation technology [IBM, 1999].

AUIML consists of two major sets of elements, those that define the Data Model, which underpins a specific interaction with a user; and those that define the Presentation Model, which specifies the "look and feel" of the UI. The Data Model is independent of the modality of user interaction and the Presentation Model allows flexibility in the degree of specificity of what exactly is expected of a renderer (a software capable of translating a AUIML document into a concrete user interface). Therefore, AUIML provides a great deal of flexibility by allowing decisions on the most appropriate concrete presentation to be taken by the renderer or to be enforced by the designer.

Figure V.8 depicts an example, from [IBM, 1999; Azevedo et al., 2000], of an AUIML document describing an interface that simply asks the user for his complete name. Figure V.9 depicts two examples of concrete UIs generated, respectively, by a Java Swing and a DHTML renderer for the AUIML document illustrated in Figure V.8.

```
<GROUP NAME="PersonName">
      <CAPTION>
            <META-TEXT>Person's complete name</META-TEXT>
      </CAPTION>
      <CHOICE NAME="PersonTitles" SELECTION-POLICY="SINGLE">
      <CAPTION><META-TEXT>Title</META-TEXT></CAPTION>
      <HINT>
            <META-TEXT>This is a set of valid titles you may choose
from.</META-TEXT>
      </HINT>
      <STRING NAME="MR">
            <CAPTION><META-TEXT>Mr.</META-TEXT></CAPTION>
      </STRING>
      <STRING NAME="MRS">
            <CAPTION><META-TEXT>Mrs.</META-TEXT></CAPTION>
      </STRING>
      <STRING NAME="MISS">
            <CAPTION><META-TEXT>Miss</META-TEXT></CAPTION>
      </STRING>
      <STRING NAME="MS">
            <CAPTION><META-TEXT>Ms.</META-TEXT></CAPTION>
      </STRING>
      </CHOICE>
      <STRING NAME="FirstName">
            <CAPTION><META-TEXT>First Name</META-TEXT></CAPTION>
      </STRING>
      <STRING NAME="Initial">
            <CAPTION><META-TEXT>Initial</META-TEXT></CAPTION>
      </STRING>
      <STRING NAME="LastName">
            <CAPTION><META-TEXT>Last Name</META-TEXT></CAPTION>
      </STRING>
</GROUP>
```

Figure V.8 – AUIML Example for a simple user-interface asking a person's complete name

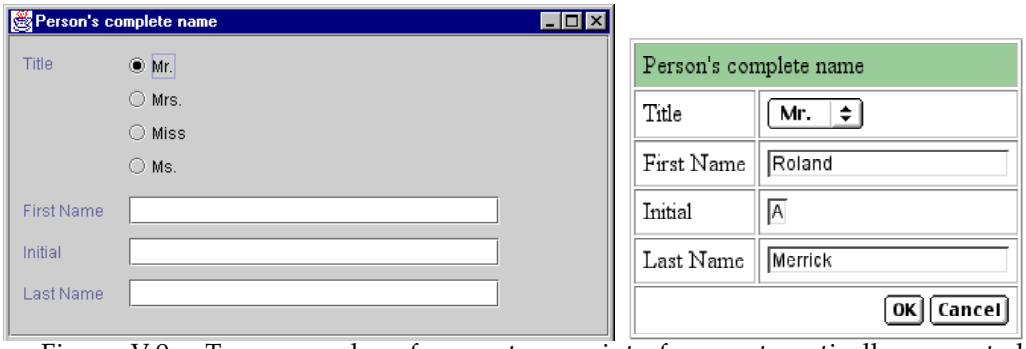


Figure V.9 – Two examples of concrete user-interfaces automatically generated from a AUIML document through: (i) a JavaSwing renderer and (ii) a DHTML renderer

The Data Model of an AUIML document structures the information necessary to support a particular user interaction. AUIML provides a set of elements for the data model as follows:

- a set of container, grouping and aggregating elements (the DATA-GROUP, GROUP, CHOICE, TABLE, ROW, TREE and NODE tags);
- a minimal set of elementary data types (STRING, NUMBER, DATE, TIME and BOOLEAN tags) that share a common set of attributes (VALID, ENABLED, READ-ONLY, MANDATORY, ENABLED, and so on) and a specification o simple event-handlers (WHEN-SELECTED, WHEN-SHOWN, WHEN-ENABLED and so on);
- a mechanism that facilitates the specification of user-defined data types (the UDT tag) and two types of complex data types (IMAGE and SOUND tags);

- a set of item manipulation elements that allow the state of other data elements to manipulated as a consequence of events (CHANGE-ITEM, REFRESH-ITEM and DISPLAY-ITEM);
- a set of user actions the user is allowed to perform on some elements of the data model. The actions can be aggregated and associated with one or more items (ACTION-GROUP, ACTION-LIST, CONTEXT-GROUP, ACTION and ACCELERATOR tags).

The presentation model in AUIML allows a considerable flexibility in the amount of specificity a designer can convey. Designers can opt to precisely control what is to be displayed or just specify the interaction style leaving the decisions to the renderer based on the underlying data model. AUIML provides a set of elements for the presentation model as follows:

- three elements enabling the specification of three common interaction techniques: wizards, property sheets and panels (the WIZARD, PROPERTYSHEET and PANEL tags). Each element implies the corresponding navigation and behavior associated with the three techniques, including common buttons and their behavior;
- a set of action containers that organize the collection of available actions, these include elements related to menus (MENUBAR, MENU, MENUITEM and SEPARATOR tags), and one element for toolbars (the TOOLBAR tag);
- a set of area constructs that which have built in layout policies for their child user interface elements, these include a set of elements that layout their contexts according to different patterns (VERTICAL-AREA, HORIZONTAL-AREA, DEFINITE-AREA, and so on);
- a number of interface elements (widgets or UI components) that can be associated with the various area tags, these include the common widgets found in any GUI (LABEL, BUTTON, TEXTFIELD, SLIDER, RADIOBUTTON, CHECKBOX, COMBOBOX, and so on).

AUIML also provides a number of common attributes (common metadata elements) to provided different degrees of information as to the purpose or function of the elements to which they apply. Those elements define three levels of information. Each of these levels has associated material, which can be used by a renderer, or application, to communicate with the user. The common metadata elements are defined in the CAPTION, HINT and TIP tags. All three of these metadata elements can contain the information in the same media oriented types. These are textual (META-TEXT tag), audio (META-AUDIO tag) and pictorial (META-IMAGE tag). The authors of the AUIML consider that over time it is desirable to move to these being containers of XHTML Semantic modules.

V.2.2.Generating AUIML Documents from Wisdom Models

The idea of generating AUIML documents from UI specific UML models, such the ones provided by the Wisdom notation, is then an attempt to assess the capability of increasing the ceiling and lowering the threshold of hypertext markup as an effective UI description language. On the one hand, the UML notation provides the low threshold of a comprehensible diagrammatic notation, while also ensuring that adequate support for semantics, support for the wider context of system development and tool interoperability. On the other hand, UI specific markup provides access to the most successful and widely used automatic generation technology available today. It supports appliance independent UI descriptions, while also ensuring that designers have flexibility and control over the design process.

The idea of exploring this approach was the motivation for a joint project involving the IBM Ease of Use group, where both the main authors of Ovid and AUIML work. Since two different UML modeling approaches were involved (Ovid and Wisdom), and the motivations were clearly different, the project collaborators decided it was beneficial to explore two tracks.

The IBM Ease of Use Group, with the active involvement of an undergraduate student of the University of Oporto, explored a pragmatic approach aiming at generating AUIML descriptions directly from Ovid UML models developed in Rational Rose. The Ovid2AUIML project aimed specifically at producing a "proof of concept" that it was feasible and useful to link Ovid models to AUIML descriptions allowing designers to immediately see visual representation of UI [Azevedo et al., 2000]. The approach followed in the Ovid2AUIML project didn't take advantage of the UML interchange formats, instead the team decided to use the scripting capabilities of the Rational Rose tool. Moreover, due to the pragmatic characteristics of this approach, little attention was paid to the semantic aspects of the additional information required in the Ovid models to generate workable AUIML documents.

Simultaneously we examined how the pragmatic approach followed in the IBM project could be generalized. Our approach aimed at exploring different possibilities of the underlying concept of generating UI descriptions from UML models. In particular, the experiences concentrated on: (i) the suitability of the UML notational extensions provided in Wisdom for the Presentation Model (see section IV.4.5) to formalize the ad-hoc parameterizations used in the IBM pragmatic approach – in particular the Wisdom stereotyped navigation and containment relationships as well as the input and output elements and the stereotyped actions; (ii) the potential of the Wisdom notational extensions to generate other hypertext markup specifications (e.g. UIML or Xforms); (iii) the impact and effectiveness of the XMI interchange format to detach the transformation process from a particular tool (see section V.1.1) (iv) the suitability and effectiveness of the eXtensible Stylesheet Language Transformations (XSLT) [W3C, 1999b] to map different XML dialects (specifically between XMI and

AUIML); and finally (v) the potential, flexibility and effectiveness of the overall process in the generalized form.

Both tracks of this project were complementary. On the one hand the IBM approach served as an effective and successful "proof of concept", as reported in [Azevedo et al., 2000]. On the other hand, our approach enabled a more in depth exploration of the envisioned potential of some technologies that are still under development and not fully supported by commercial tools (e.g. XMI, XSLT and AUIML). Figure V.10 depicts an overview of both tracks in the UML to AUIML project. To the left side is the IBM approach and to the right side is our approach, together with an overview of the transformation process between Wisdom UML models and AUIML.

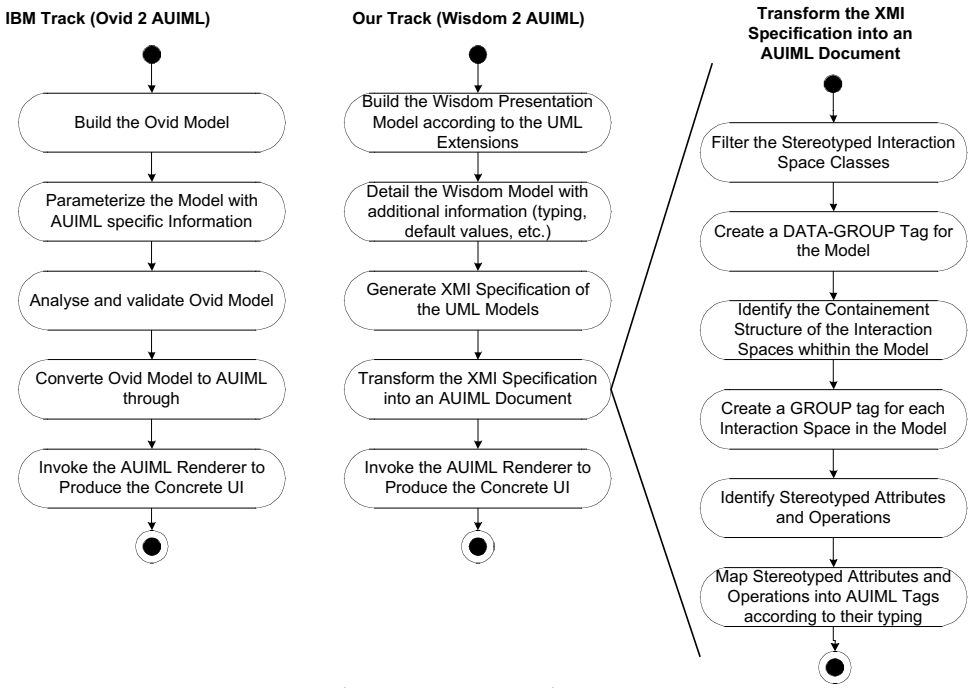


Figure V.10 – Overview of the two tracks in the UML to AUIML project, including an overview of the XMI to AUIML transformation process

Both approaches, described in Figure V.10, start with building an abstract object model of the presentation aspects of the envisioned interactive application (see sections III.3.4, III.4.2.1 and IV.4.5). From this initial UML specification, both approaches differ significantly in the process by which the end AUIML document is produced. While the IBM approach uses the scripting facilities provided by the modeling tool (Rational Rose) to directly generate the AUIM document, our approach uses the XMI interchange format. Therefore, the transformation process, in our approach, involves producing the XMI specification and then applying a set of XSL transformations (XSLT) to the XMI document, in order to map the UML constructs into AUIML elements. Both the XMI generation and the XMI to AUIML transformation processes are fully automated. The mapping between the UML modeling constructs, defined in the Wisdom presentation model, and the AUIML elements is described in the right-hand side of Figure V.10. The transformation process involves filtering the XMI file for all the interaction space class stereotypes

and the corresponding containment relationships. From this information an AUIML GROUP element is created for each interaction space in the Wisdom presentation model. The containment structure in the AUIML document is derived by the different containment relationships defined between the interaction spaces in the Wisdom presentation model (see section IV.4.5). Each AUIML GROUP element is then augmented with additional information derived from the stereotyped attributes (input and output elements) and operations (actions) defined in the Wisdom presentation model for each interaction space class. This further mapping depends on the high level typing of the stereotyped input elements, output elements, and actions (see section IV.4.5). For instance, an input element of type choice is translated into an AUIML CHOICE element, and the same applies for STRING, NUMBER, DATE and so on. Given that a designer is aware of those elementary data types, no additional information, or knowledge about AUIML, is required to produce a Wisdom presentation model capable of generating a working AUIML document. The automatic generation process ends invoking the renderer to produce a concrete user interface from the AUIML document.

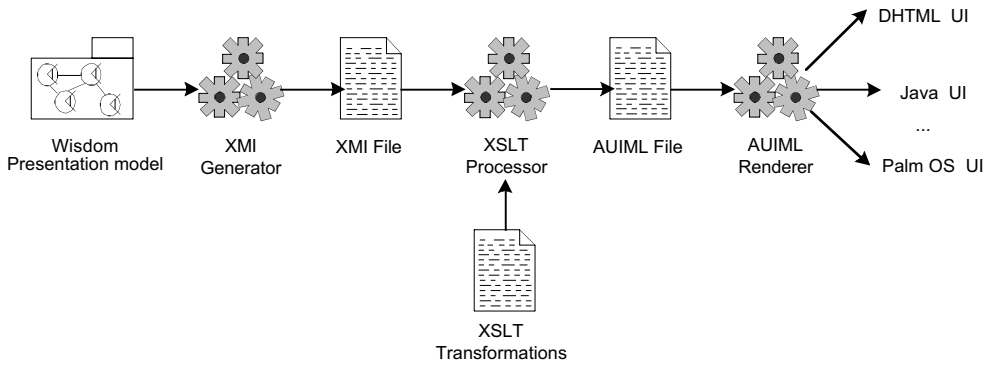


Figure V.11 – Overview of the Wisdom to AUIML Transformation process

Figure V.11 depicts the transformation process explored in our approach. This process is a fully automated; given that designers supply additional types for input elements, output elements and actions for each interaction space in the Wisdom presentation model. Therefore our experiences confirmed that it is possible to generate multiple concrete user-interfaces from an abstract UML-based specification of the interaction. Moreover, that process is fully automated and independent of the modeling tool used to produce the UML models (given that the tool supports the XMI standard).

An example of the Wisdom to AUIML automatic generation process is provided in Figure V.12 and Figure V.13. The first example shows the Wisdom presentation model for customer information in the context of a hotel reservation system – a similar *manual* example is provided in Figure IV.27. As we can see from the UML model in Figure V.12, this abstract model of the user interface comprises three interaction spaces: one for the overall customer information, one for the customer address and one for the customer preferences. Each interaction space includes further presentation information in the form of different input and output elements and actions. To the left of Figure V.12 is the XMI document attained by exporting automatically exporting the

UML model in the modeling tool. XMI document contains information for the Wisdom specific stereotypes and also the information about the model itself. By traversing the XMI document through the XMI.id of each class and stereotype, we can isolate the Wisdom presentation model and extract the specific interaction spaces and relationships (contains also stereotyped). The same procedure applies to the stereotypes attributes (input and output element) and operations (actions). Therefore a set of XSL transformation can provide the mapping that translates this XMI document into the AUIML document illustrated in Figure V.13. As we can see from this example it is possible to fully generate an AUIML document from a Wisdom presentation model in XMI only applying XML based technology (XSL transformations). The process is fully automated by an XSLT processor and requires no manual manipulation of the information. Furthermore, taking advantage XMI validation (see section V.1.1.3 and Figure V.5) we can verify that both the XMI document and the resulting AUIML document are well-formed and comply to the syntactical and semantic requirements defined in the corresponding Document Type Definitions (DTDs).

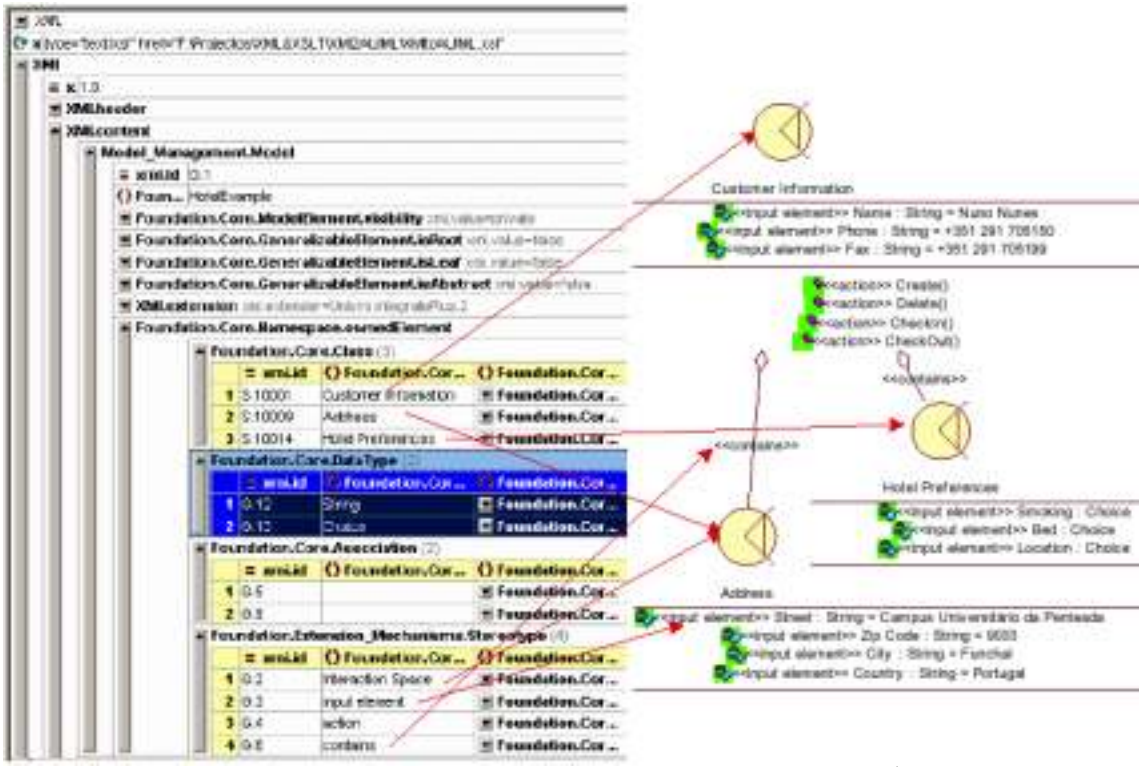


Figure V.12 – An example of a Wisdom presentation model and the corresponding XMI document

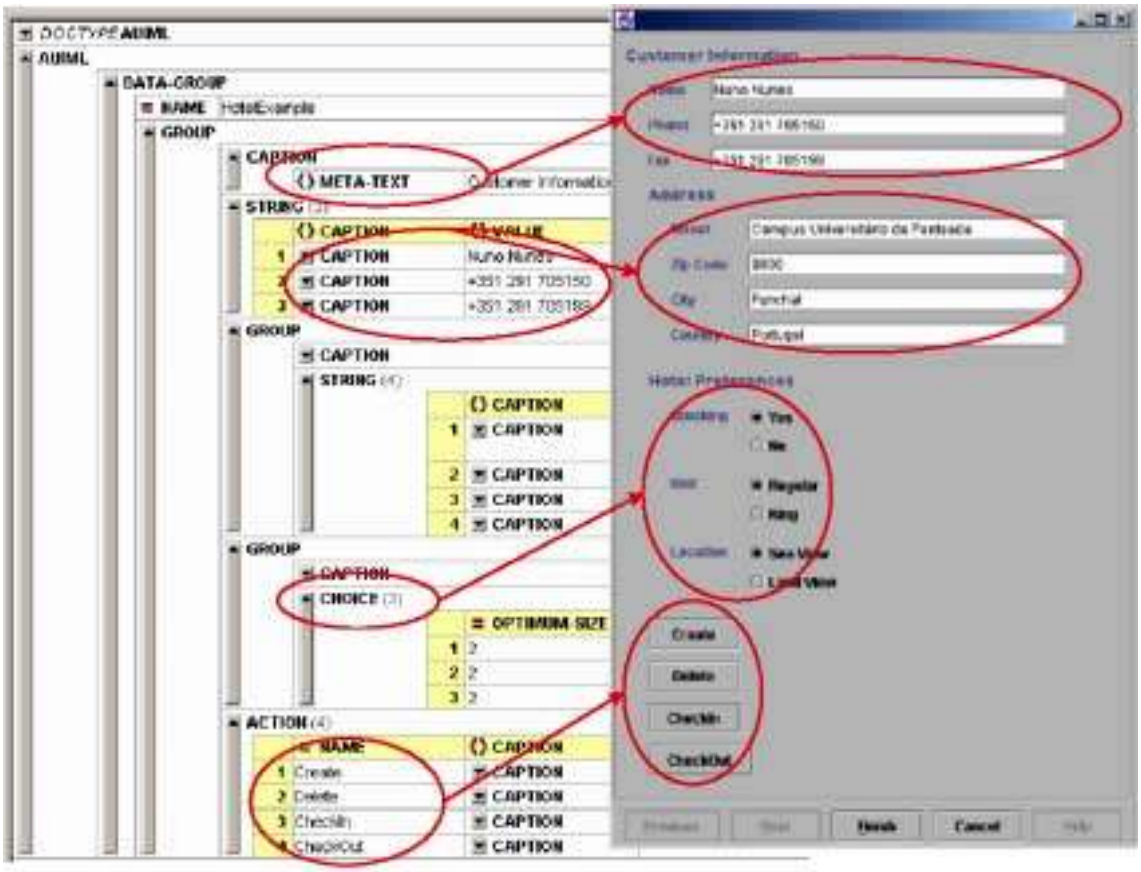


Figure V.13 – An example of an AUIML document and a corresponding concrete user interface produced by a Java Swing renderer

The experiences with this generalized Wisdom to AUIML transformation were conducted at a highly experimental level. This was mainly because most of the underlying technologies are still incipient and not fully supported by the tool vendors. Although XMI is established as a standard for a reasonable amount of time [OMG, 2000], it is still not widespread as a de facto interchange format. The only UML modeling tool that we had access to, which supported XMI was Rational Rose 2000 [Rational, 2000]. Nevertheless, Rose still requires the support of a third party plug-in, which doesn't fully support XMI 1.0 [OMG, 2000]. Moreover, XML processors that fully support XSLT [W3C, 1999b] are not widely available in current desktop operating systems. Finally, AUIML is still an internal working specification within IBM, therefore the existing renderers are still under development and don't fully support the AUIML specification.

Despite all of the limitations with the underlying standards and technologies, the Wisdom to AUIML project proved that the notational extensions provided in the Wisdom presentation model (see section IV.4.5) are powerful enough to drive an effective automatic generation process. The concept of an interaction space, including input and output elements and actions, proved to be a simple but effective way to encapsulate a more complex appliance independent user interface description language such as the AUIML. The UML extensions enable the encapsulation of AUIML specific markup, therefore obviating the need for designers to learn and

manipulate another language and notation. Moreover, they are easier to manipulate and relate to the wider context of interactive system development – as it is implied in the additional Wisdom models (see section IV.4). The experiences also proved the effectiveness of the XMI interchange format to eliminate tool dependency and simplify the interchange of modeling information between tools. Not only the information in the models can be exchange, but also the meta-modeling extensions (stereotypes, tagged values and constraints), thus enabling approaches like Wisdom to take advantage of extensibility mechanism without loosing the power of the underlying technologies. One such example was the ability to take advantage of the XSL transformation to perform the mapping between the two XML-based dialects, thus obviating the need to develop specific translators.

There are also limitations with the approach described here for automatic generation of AUIML documents from Wisdom presentation models. The first limitation is concerned with the different mappings between the Wisdom modeling constructs and the AUIML elements. For instance, interaction spaces are always translated into AUIML GROUPs, but there are situations where other elements would be more appropriate (in particular TABLE and ROW, TREE and NODE). Moreover, our approach only considered the generation of the AUIML data model, i.e., no information is generated for the AUIML presentation model. The motivation for this restriction is related to the fact that the Wisdom presentation model is a highly abstract model of the user-interface and it doesn't include any information regarding the specific interaction styles or widgets (see section IV.5.5). However, it is possible to include this kind of information using UML tagged values attached to the Wisdom specific UML constructs. A UML tagged value is a standard extensibility mechanism that permits arbitrary information to be attached to any modeling construct [OMG, 1999]. Tagged values are usually applied to attach information that doesn't change the underlying semantics of the models, for instance tool specific information for back end purposes (code generation, reporting, and so on). Since the interaction style, techniques and widget specification don't change the semantics of the Wisdom presentation model, UML tagged values are the best way to include additional information to enforce the generation of elements appearing in the AUIML presentation model. For example, in Figure V.13 the renderer decided to present the customer information with a wizard interaction style, since this is obviously not the most adequate choice, the designer could attach a tagged value to the topmost interaction space enforcing a different interaction style. The same mechanism could be used with input elements, output element and actions, to enforce rendering with specific widgets.

In the next section we discuss how this experience could be generalized to the forthcoming W3C standard XForms. We also envisioned how the principles explored here could leverage a new generation of UI tools that combine the power of automatic generation and the flexibility of graphical editors.

V.2.3.Generalization: XForms and Flexible Automatic Generation

XForms are the W3C answer to the increasing demands of web applications and e-commerce solutions to implement the interaction between the user and the system. With the advent XHTML [W3C, 2000b], the World Wide Web Consortium (W3C) successfully translated the basic capabilities of the HTML 4.0 into an XML-validatable series of modules. This initial goal enabled the W3C to explore new possibilities, introduced by the XML, for different areas. One of the areas concerned the need to overcome the limitations of the current design of web forms that doesn't separate the purpose from the presentation of a form. This effort started within the W3C as a subgroup of the HTML Working Group but recently spun off as an independent XForms Working Group. The discussion of XForms presented here corresponds to the latest W3C working draft of 16 Feb. 20001 [W3C, 2001], which is "work in progress" provided by the W3C working group for review by W3C members and other interested parties. Many sections of the working draft described here are incomplete and subject to change by the W3C.

XForms is the W3C's name for a specification of Web forms that can be used with a wide variety of platforms including desktop computers, handhelds, information appliances, and even paper [W3C, 2001]. The key goals of the XForms are [W3C, 2001]:

- Support for handheld, television, and desktop browsers, plus printers and scanners;
- Richer user interface to meet the needs of business, consumer and device control applications;
- Decoupled data, logic and presentation;
- Improved internationalization;
- Support for structured form data;
- Advanced forms logic;
- Multiple forms per page, and pages per form;
- Suspend and Resume support;
- Seamless integration with other XML tag sets.

XForms are comprised of separate sections that describe what the form does, and how the form looks. This allows for flexible presentation options to be attached to an XML form definition (including classic XHTML forms). The diagram at the left hand-side of Figure V.14 illustrates how a single device-independent XML form definition (the XForms Model) can work with a variety of standard or proprietary user interfaces. The XForms User Interface provides a standard set of visual controls that are targeted toward replacing the existing XHTML form controls. These form controls are directly usable inside XHTML. It is expected that other groups within the W3C (such as the Voice Browser Working Group), other companies and standard organizations can

independently develop user interface components for XForms. Another important concept in XForms is that forms collect data, which is expressed as XML instance data in the right hand side of Figure V.14. To support this process a channel of instance data is required to enable the flow of instance data to and from the XForms processor. The XForms Submit Protocol defines how XForms send and receive data, including the ability to suspend and resume the completion of a form [W3C, 2001].

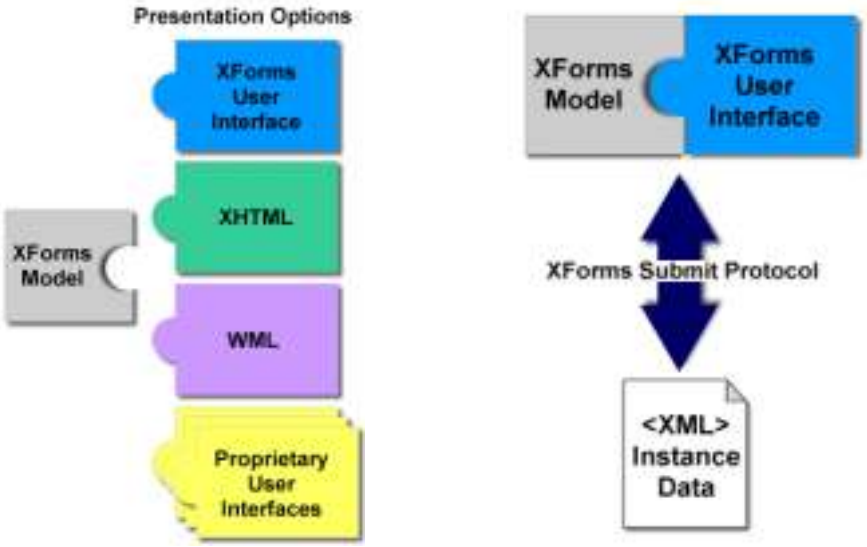


Figure V.14 – XForms as an appliance independent user-interface description language (source: [W3C, 2001])

The XForms Model is, therefore, the non-visual definition of an XML form. The XForms Model comprises two components: Data structures and XForms Extensions. The Data Structures component provides a schema describing the instance data and providing reusable data types. The XForms Extensions include aspects that are not typically expressed in schemas, like additional constraints, dependencies and Dynamic Constraints Language, and calculations [W3C, 2001]. The aim of this definition of the XForms Model is to support aspects that are not typically addressed in schemas, examples include for instance: stating that an item is read-only; defining complex interrelations between model items (a spouse information section that only applies if the user indicated that he has a spouse); numerical calculation and representation of numeric values using decimal arithmetic; workflow, auto-fill and pre-fill form applications.

To some extend, the XForms Model intend at modeling the information for presentation purposes in the same way that Wisdom interaction spaces model the user-interface (see section IV.4.5). Moreover, the XForms Data Structures are similar to the elements described in the AUIML data model (see section V.2.1.1), that is, they encompass simple datatypes (string, boolean, number, currency, date, time, binary, etc.), compound datatypes and facets (single defining aspects of a value space, for instance, enumeration, length, max, min, etc.) [W3C, 2001].

The XForms User Interface defines the part of the electronic form that is perceivable by users, through a set of visual controls that are targeted towards replacing the existing XHTML controls. Typically XHTML form controls are expressed in fairly generic terms, for instance <select> to represent menu controls [W3C, 2001]. The XForms User Interface aims at providing increase flexibility and control over those presentation elements. XForms enable this flexibility by providing multiple specific presentations (for different types of appliances) to be attached to a single XForms Model through a flexible binding mechanism (as illustrated in Figure V.14) [W3C, 2001]. Furthermore, the XForms User Interface provides a set of XHTML modularized elements to define widgets (form controls). This set of form controls is designed for use with XHTML, and initially inspired in the existing HTML 4.0 controls. However, they eliminate the need for scripting in detriment of declarative markup.

The XForms User Interface is also similar to the elements in the AUIML presentation model described in section V.2.1.1. XForms controls are declared using markup elements and their behavior is refined via markup attributes. In addition, they can be *decorated* with style properties that can be set using Cascading Style Sheet (CSS). The set of XForms controls is divided into:

- abstract form controls - controls that share similar properties such as caption, hint, help, etc. These correspond roughly to AUIML common metadata elements (section V.2.1.1);
- core form controls – a set of controls inspired in the HTML 4.0 controls, such as output, text entry, checkbox, single select (radio buttons, drop-down menus and list boxes), multiple select (lists), buttons and submit (for submitting instance data). These controls correspond roughly to the interface elements in the AUIML presentation model (section V.2.1.1);
- custom form controls – a mechanism for allowing the creation of reusable and custom user interface components. There is no mechanism for custom form controls in AUIML;
- multiple pages – a section still under development for allowing multiple page display. Mutiple pages are also not supported in the AUIML;
- layout – a set of elements that allow control over the presentation technology, such as elements for grouping, additional style properties, grid layout, text and graphics. These correspond to area constructs in the AUIML (see section V.2.1.1).

From the above description of the XForms User Interface, and even considering that this is work in progress subject to changes until the recommendation stage is reached, it seams obvious that the experiences described in section V.2.3 could be easily replicated for XForms. The rationale for such an approach would follow the same mapping principles described in section V.2.2 for automatically translating the Wisdom presentation model to the AUIML. Such a mapping, under the light of the current XForms working draft, would comprise the following:

- Mapping Wisdom stereotyped interaction spaces into XForms Model group elements that allow aggregate hierarchical arrangement of ordered datatypes (including other group elements). Through tagging Wisdom interaction spaces the mapping could be enforced for different XForms Model elements, such as array and union;
- Mapping Wisdom stereotyped containment relationships between interaction spaces into the hierarchy of XForms Model group elements, for instance, if interaction space A has a contains association with interaction space B, then group element B is nested in group element A;
- Mapping Wisdom stereotyped input elements and output elements into XForms datatypes, using conceptual typing of the stereotyped attributes, for instance, typing input and output elements as string, boolean, number, currency, monetary, date, time, duration, URI, binary, enumerated. Output elements would additionally have the model item property read only;
- Mapping Wisdom stereotyped actions into XForms User Interface buttom elements, which may contain an XForms Dynamic Constraint to call when the control is activated. The current XForms working draft is not specific regarding the description of actions in the XForms Model, in which case it would be beneficial to map Wisdom stereotyped actions into such an XForms Model element;
- Mapping Wisdom stereotyped navigation relationships into the navigation sequence defined for XForms controls deriving from the abstract form control anyNavControl, through the navindex attribute;
- Tagging different Wisdom modeling elements (through UML tagged values) to enforce mapping to presentation specific elements in the XForms User Interface. Examples include, forcing specific binding of XForms Data Model elements (datatypes) into specific XForms controls.

The approach described before, supported by the concrete experiences conducted with IBM's AUIML described in section V.2.1.1, reinforce the potential of UML model-based specification of appliance independent user interfaces. On the one hand, the UML based extensions proposed in the Wisdom presentation model leverage on increasing support for the UML standard, both in terms of tool support and interoperability, but also, and foremost, because it is possible to use the same modeling to specify the user interface and the internal system functionality. On the other hand, software engineering can incorporate many contributions from the usability-engineering field, in particular at this level support for automatic generation of user interfaces and tool support for user interface design.

As we discussed in sections V.1.2.2 and V.1.3, automatic generation of user interfaces is an established technology in the WWW that benefited from a low-threshold and a low ceiling, due to the limitations of current web forms. The existing web forms combine three successful UI-related technologies: simple but limited appliance independent UI description language (HTML forms), event-based scripting and end-

user customization through CSS (see sections V.1.2.2 and V.1.3). All of those technologies addressed an important, but limited part of the problem with a low ceiling and a low-threshold approach. XForms are an important step towards increasing the ceiling of web-based UIs, but will eventually fall into the same problems as conventional model-based design of UIs (see sections II.7 and V.1.2.2). The basic problem that XForms face is that increasing the support for richer interaction will also increase the threshold of the underlying UI description language (including increased complexity for the automatic generation process and limiting the control of designer over the UI).

We claim that the Wisdom specific extensions can be used to lower the threshold of appliance independent user interface description languages (like the AUIML or XForms), because they provide designers with a diagrammatic language that can be used to generate a *first cut* into the presentation model. This initial specification, on the one hand, is easily related to other modeling artifacts required to convey all the aspects of an interactive application (see section IV.3); on the other hand, it can be used to generate effective user interfaces as we demonstrated in section V.2.2. To support the flexibility required to produce high quality user interfaces, a graphical exploration tool could be used to manipulate the presentation specifics, the layout and other decorative elements. This approach is illustrated in Figure V.15, which is an elaboration of the experimental environment described in the Wisdom to AUIML project (see section V.2.2 and Figure V.11).

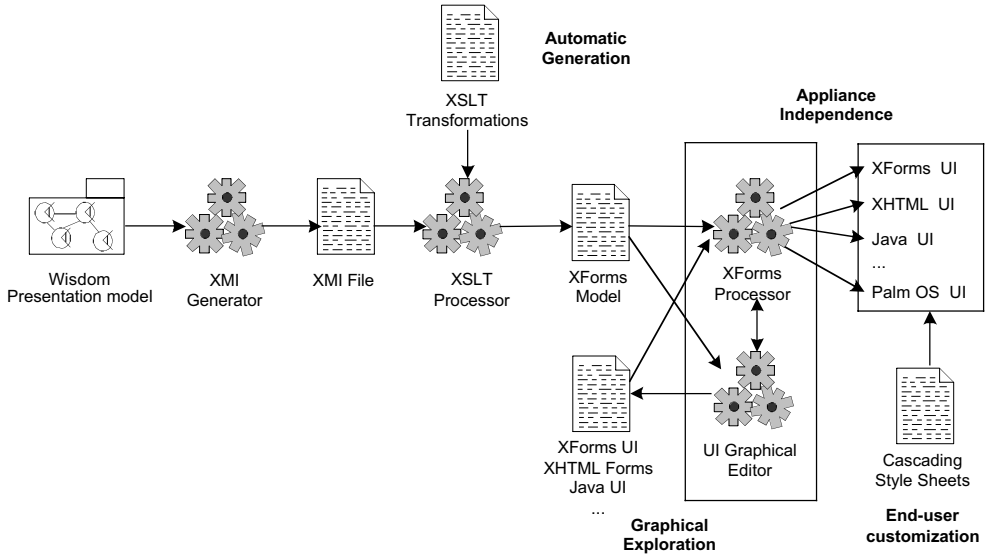


Figure V.15 – An example of an XForms-based flexible automatic generation process through graphical exploration

The flexible automatic generation process depicted in Figure V.15 combines several successful technologies and limits each one of them to a particular part of the UI design process. The Wisdom presentation model describes the UI in terms of abstract implementation independent entities that are structure to realize the interaction with the user (see section IV.3.2). The different interaction spaces and the containment and navigation relationships among them define the Wisdom presentation model. Using

the Wisdom UI architecture (see section IV.3.2) the interaction spaces are related to the other entities (boundaries, entities, controls and tasks) that define the overall architecture of the software system. At this level designer's concentrate on the overall system architecture and the architectural significant presentation elements, for instance, how information and actions are grouped to support the user in their tasks. At the design level designer detail each interaction space in terms of the different input elements, output elements and actions that each interaction space requires to support a task or a set of interrelated tasks. With adequate typing of those stereotyped attributes and operations designers are able to produce an initial abstract and device independent model of the specific user interface. That model can be translated and rendered into a concrete user interface using automatic generation techniques such as the ones described in section V.2.2 (or the ones envisioned in this section for XForms). Additional UI specifics can be manipulated over the automatically generated user interface (the XForm Model) through a graphical exploration tool that produces the specific UI model (XForms UI, XHTML Forms, or any other specific UI). The final user-interface is then accomplished using an XForms processor (or an AUIML renderer) that enables rendering into the target platform (browser or any other mechanism). The model described here and depicted Figure V.15 can also take advantage of the XMI. Since XMI allows the information to be preserved and passed through a tool that does not understand the information (see section V.1.1.2), the UI specifics can be encoded via UML tagged values and attached to the XMI file for additional manipulation by other tools. One example of such tool interoperability is discussed in the next section.

V.3. ENHANCING THE SUPPORT FOR TASK MODELING WITH TOOL INTERCHANGE

Task modeling is recognized to be a central part of the user interface design process (see section III.3.1 and [Bomsdorf and Szwillus, 1998; Paternò, 2000]). Modeling the user's task enables designers to construct user-interfaces that reflect the task properties, including efficient usage patterns, easy-to-use interaction sequences, and powerful assistance features [Bomsdorf and Szwillus, 1998]. Most of those benefits come from adequately performing the transition from task (actual tasks) to dialogue (envisioned tasks), in Wisdom terms from the essential task flows (IV.2.1.2) to the dialogue model (IV.4.4). Once the decomposition of the envisioned task model reaches the basic tasks, which involve no problem solving or control structure (actions as defined in section II.1.2.2), the task model actually reflects the dialogue model that fully describes how the user interacts with the system. If such a level of description is reached it is possible to directly implement the task model in the final system to control the dynamic enabling and disabling of the interaction objects and to support features, such as context-dependent task-oriented help [Paternò, 2000; Paternò, 2001]. However, this vision of task modeling highly depends on tool support. When supported by adequate tools dialogue models can be used to perform evaluation, user interface generation, context sensitive help, consistency and completeness checks and design assistance in exploring the user interface design space (see section III.2.3).

As a result of a workshop about task-based user interface design [Bomsdorf and Szwillus, 1998] the participants defined a set of issues that tools should support, as follows:

- Flexible and expressive formalisms that are able to capture most relevant aspects of an interface design;
- Hiding or conveying the formalism in a unobtrusive manner that yields a rather natural outgrowth of tools to support various stages of design;
- Flexible automation, meaning that tedious or mechanical aspects are automated but creative aspects are flexibly controllable enabling designers to explore different alternatives for design.

In section III.4.3.2 we discussed different alternatives to support task modeling in the UML. One of the justifications for adapting the CTT formalism in the Wisdom dialogue model was to take advantage of one of the most successful and widely used notation in the HCI field (see sections II.7 and IV.4.4). CTT was developed taking into account previous experience in task modeling and adding new features in order to obtain an easy-to-use and powerful notation. As Paternò points out "the key issue is

not only a matter of expressive power of notations but it is a matter of representations that should be effective and support designers in their work rather than complicate it" [Paternò, 2001]. Moreover, tools support for task modeling is recognized to be insufficient [Bomsdorf and Szwillus, 1999], and CTT is one notation that has extensive support for task modeling, consistency checking and simulation [Paternò, 2000]. In this section we describe several experiences where we used the XMI interchange format to take advantage of the existing CTT tool support without requiring manual re-entry of information.

As we discussed in [Nunes and Cunha, 2001a], the different experiences carried out encompass using two XSL style sheets to transform the Wisdom dialogue model into the CTT XML schema and vice-versa. This process enables user interface designers or HCI experts to use the powerful CTT notation, and the associated tools, while being able to change artifacts with their software development counterparts. The transformation process relies on the fact that the CTT tools support-expressing CTT in the XML format [Paternò et al., 2000].

This UML transformation is briefly outlined in Figure V.16 and Figure V.17. The example used here is from a check availability task in a hotel reservation system described in section IV.5.5.

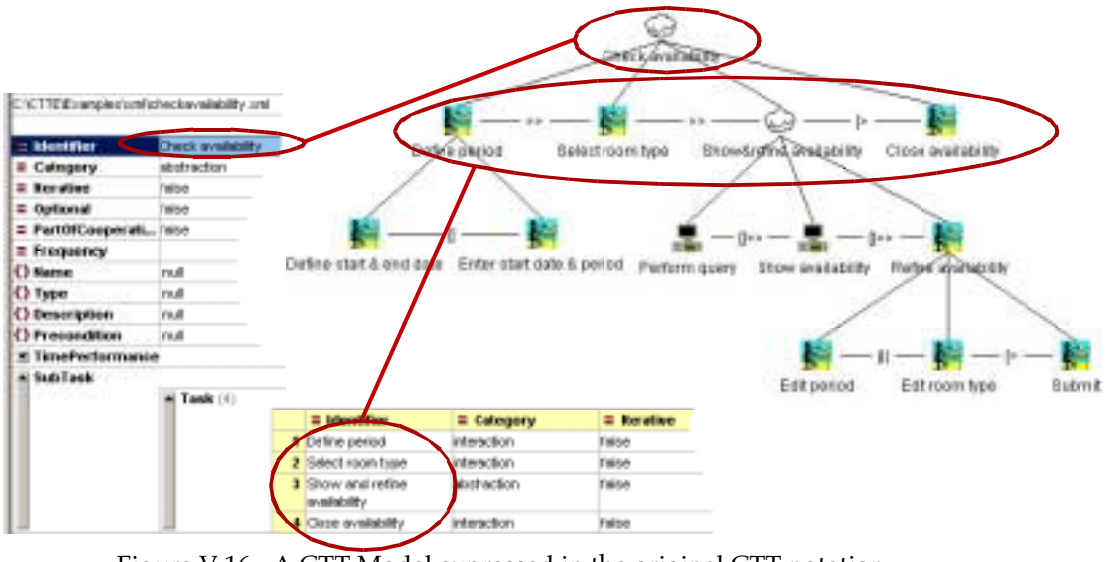


Figure V.16 - A CTT Model expressed in the original CTT notation

Figure V.16 illustrates a CTT task model in both diagrammatic and XML formats. The diagrammatic model, at the far right, was produced using the CTTe tool. At the left of the figure is the corresponding XML specification generated by the CTTe tool and later opened for browsing in a popular XML editor. As we can see from the red highlighting, the hierarchical structure of the task tree is built using a sequence of <task> <subtask> XML tags. Although it's not visible in the figure, the temporal relationships are specified with the following tags per subtask: <Parent name="Parent Task"/> <SiblingRight name="subtask"/>.

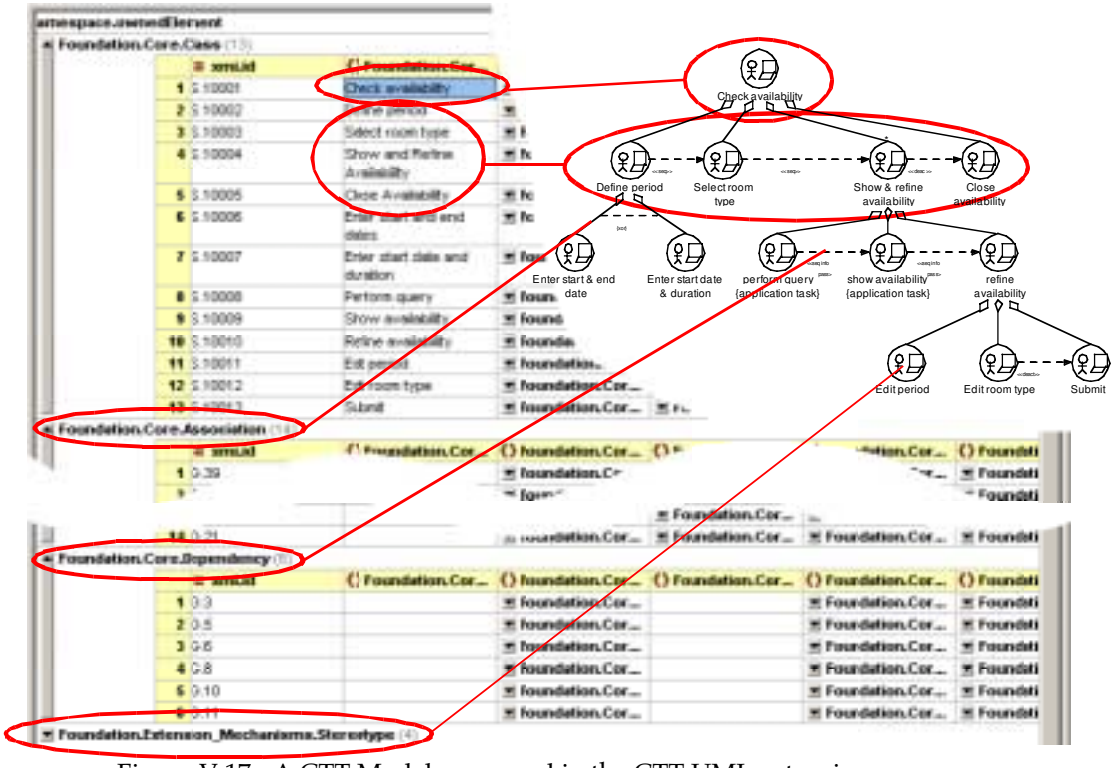


Figure V.17 - A CTT Model expressed in the CTT UML extension

Figure V.17 illustrates the same task model expressed with a UML class diagram (at the upper right) and the corresponding XMI specification (see section V.1.1.3). As we can see from the figure, the XMI schema is significantly different form the CTT XML schema. The different classes in this example are under the XML tag <Foundation.Core.Class>; the hierarchical structure is captured with association stereotypes and is under <Foundation.Core.Association>; and finally the temporal relationships are captured with dependency stereotypes under <Foundation.Core.Dependency>. In addition all the stereotypes for the core elements are under <Foundation.Extension_Mechanisms.Stereotype>. This way the XSL style sheet must lookup the different stereotypes from the corresponding section and them filter the different modeling components. The hierarchical decomposition of tasks is reconstructed following the aggregation relationships and the temporal relationships generated from each level of subtasks.

The approach described before enables designers to take advantage of existing modeling, eliciting and simulation tools for the CTT task notation; while supporting automated artifact change with UML tools and developers of the internal software functionality that supports the end user tasks. Moreover, task models are useful to complement the automatic user interface generation process described in sections V.2.2 and generalized in section V.2.3. The automatic generation process described in Figure V.15 can benefit from information conveyed in the task model. That information is usually associated with a number of criteria conveyed in the task model that is important do identify interaction spaces and guide the presentation model in

order to obtain effective user interfaces. The criteria described in detail in section IV.5.5, can be summarized as follows [Paternò, 2001]:

- grouping of tasks can be reflected in the grouping of interaction spaces;
- temporal relationships among tasks can be useful to structure the user interface dialogue and indicate when the interaction techniques supporting the tasks are enabled;
- the type of tasks and interaction spaces (and their cardinality) should be considered when the corresponding widgets and presentation techniques are selected.

Although we didn't evaluate the possibility of using the task model to inform the automatic generation process, an envisioned environment where our experiences interchanging task model between UML modeling tools and CTT tools are combined with that possibility for automatic generation purposes is depicted in Figure V.18

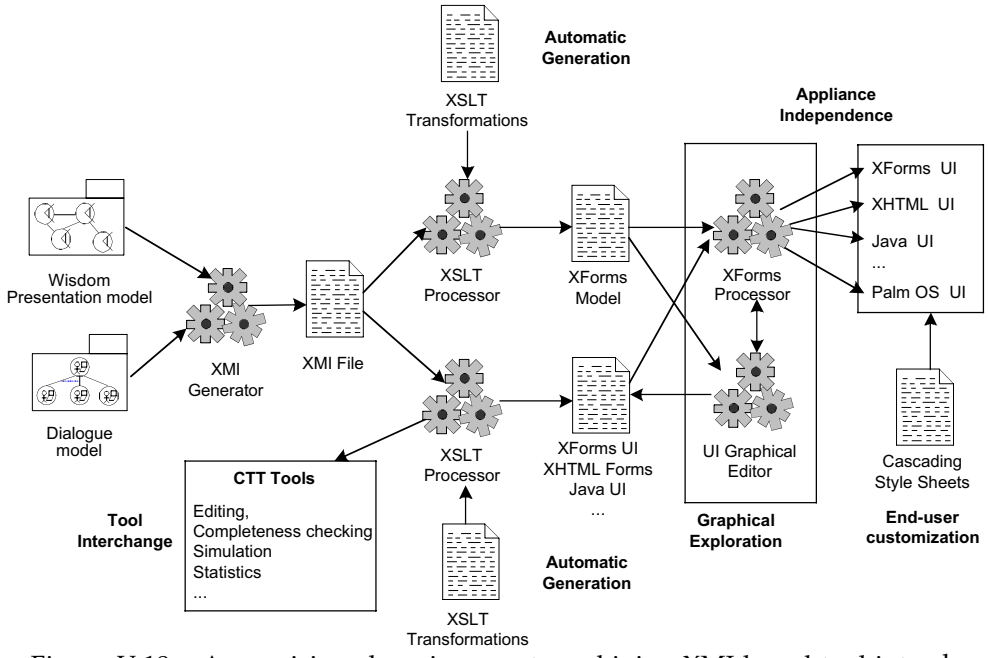


Figure V.18 – An envisioned environment combining XMI-based tool interchange and automatic generation informed by the dialogue model

V.4. THE WISDOM NOTATION AND USER-INTERFACE PATTERNS

There is a growing interest in the possibility of using patterns [Alexander et al., 1977] in user interface design, development and evaluation [Rijken, 1994; Bayle et al., 1998; Tidwell, 1998a; Paternò, 2000; Welie and Troedtteberg, 2000]. As we discussed in section II.4.3.2, patterns emerged from the ideas of the architect Christopher Alexander and are used to systematize important principles and pragmatics in the construction field. Those ideas have inspired the object-oriented community to collect, define and test a variety of solutions for commonly occurring design problems [Gamma et al., 1995]. User interface patterns follow the same principle defined by Alexander (see section II.4.3.2), a user-interface pattern is thus a general solution for commonly occurring design problems in interactive system development.

Although the interest in user interface patterns started in the mid 90s [Rijken, 1994], and even though several collections of patterns are published and publicity available [Tidwell, 1998b; Perzel and Kane, 1999; Bayle et al., 1998], a consensual pattern language as not yet emerged [Welie and Troedtteberg, 2000]. There appears to be a lack of consensus over the format and focus for user-interface patterns. In [Welie and Troedtteberg, 2000] the authors argued that user-interface patterns should focus on the usability aspects that primarily benefit users. The point behind Welie and Troetteberg's point is that several solutions in user-interface design solve problems that designers (and other stakeholders) have but that don't necessarily benefit users (for instance a banner in a web page) [Welie and Troedtteberg, 2000]. This understanding of user-interface patterns is clearly consistent with Alexander's original definition and the subsequent variation in software patterns [Gamma et al., 1995; Fowler and Scott, 1999]. In addition, several authors proposed different formats to represent user-interface patterns. The different formats proposed also follow the initial ideas of Alexander, in particular the description of a UI pattern usually encompasses the following attributes [Alexander et al., 1977; Gamma et al., 1995; Paternò, 2000]:

- Identification - including classification and other well-known names for the same pattern;
- Problem - including intent, motivation, applicability and usability problems addressed. The description of the problem usually encompasses concrete applicability examples of the pattern, for instance, scenarios, screenshots, etc.;
- Solution – including a descriptions of the elements that makeup the pattern. The solution doesn't describe a particular design or implementation because the

pattern can be applied in many situations, instead the pattern provides an abstract description of a design problem and how general arrangement of elements solve it;

- Consequences – including results and tradeoffs of applying the pattern and relationship to other patterns (reference to related patterns, variants and sub-patterns).

The attributes for identification, problem and consequences are generally described through natural language or concrete artifacts for the case of depicting the problem. However, the main problem describing patterns is the possibility of depicting the solution in an abstract way that promotes reuse in many analogous, yet different situations. The possibility of presenting that information through object-oriented notations, such as in [Gamma et al., 1995] was ultimately important for the success of analysis [Fowler, 1996] and design patterns [Gamma et al., 1995]. Object-oriented notations, such as the UML, enabled the identification and dissemination of software patterns, because developers had access to a comprehensible notation to depict abstract representations that they could instantiate during implementation.

We argue that one of the problems preventing the identification and dissemination of user-interface patterns is the lack of a modeling notation capable of illustrating in an abstract way the solution that those patterns convey. Looking in the literature in user-interface patterns we find the solution depicted either through concrete examples (for instance screenshots) or through textual descriptions or ad-hoc sketches [Tidwell, 1998a; Perzel and Kane, 1999; Bayle et al., 1998; Welie and Troedtteberg, 2000]. Moreover, those descriptions of patterns usually focus on the presentation aspects and neglect the structure of the task underlying the pattern. One notable example, in the other extreme, is provided in [Paternò, 2000] where the solution for task patterns is depicted using the CTT notation, thus being both abstract and concentrating in the task structure. However, Paternò's approach excludes the presentational aspects of the pattern, which are ultimately important for designers.

The Wisdom notation provides notational constructs for depicting both the task and presentation aspects of patterns. The Wisdom notation for the dialogue model (see section IV.4.4) is based on CTT, so a clear mapping exists from the task patterns provided in [Paternò, 2000]. Furthermore, the Wisdom presentation model, notably interaction spaces (see section IV.4.5), provides a means to convey abstract presentation aspects of conventional user-interface patterns. Thus it is possible to represent in an abstract way a set of already identified patterns, such as the ones provided in [Tidwell, 1998b; Perzel and Kane, 1999; Bayle et al., 1998; Welie and Troedtteberg, 2000].

Figure V.19 exemplifies how the Wisdom notation for the dialogue and presentation models can be used to illustrate the Wizard pattern from [Welie and Troedtteberg, 2000]. This pattern solves the problem of a user wanting to o achieve a single goal, which involves several decisions that are not completely known to the user. This

particular pattern is widely used in many applications (for instance Microsoft Office applications) to support infrequent tasks that involve several subtasks where decisions need to be made. The Wisdom dialogue model to the right hand side of Figure V.19 defines a minimal sequence of tasks, with the corresponding temporal relationships, that support an abstract Wizard. To the right-hand side of the figure is a Wisdom presentation model that illustrates how an abstract Wizard can be modeled through two interaction spaces, one for the wizard body and another one for each wizard step. Abstract actions are associated with each interaction space denoting typical actions performed in a Wizard pattern. As we can see from the example, both the dialogue and presentation models illustrate the pattern without committing to a particular design, implementation technology, platform or interaction technique.

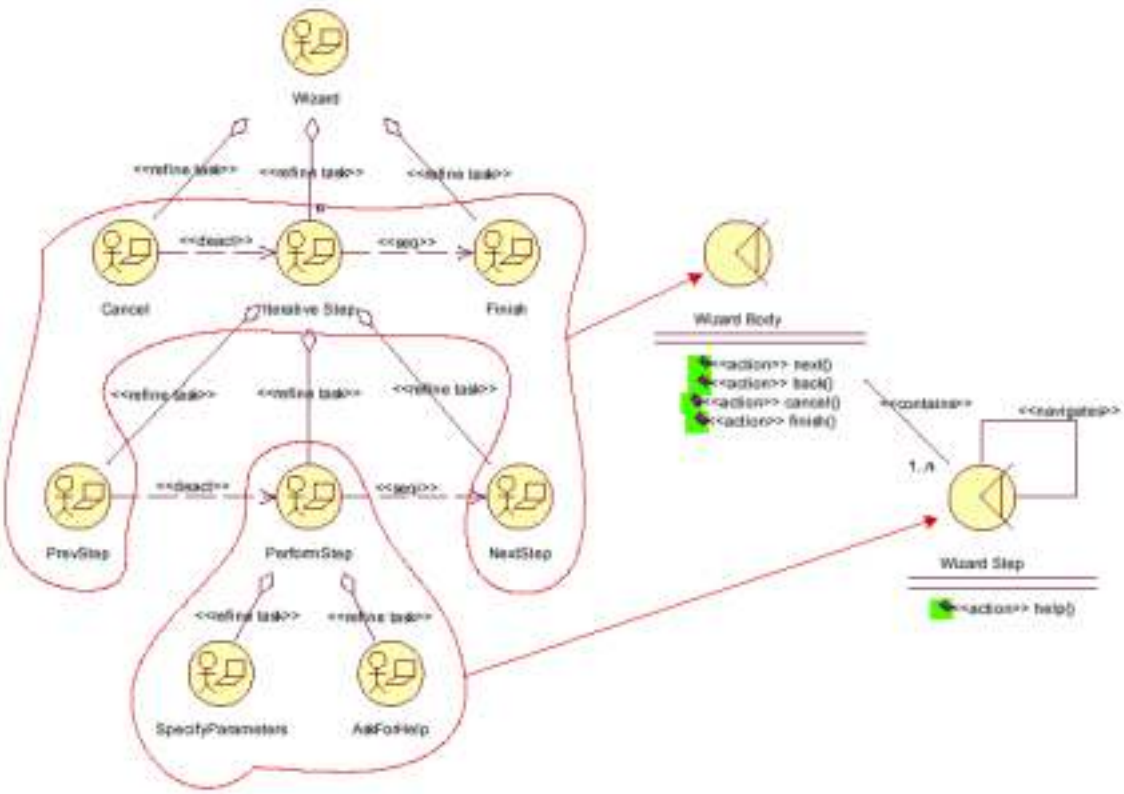


Figure V.19 – A solution for the Wizard pattern using Wisdom presentation and dialogue notations

User-interface patterns are a very important step towards promoting consistency in user-interfaces. User-interface design is becoming a more complex and demanding task (see section V.1.2.2) with the advent of multiple information appliances. Therefore, the capability of identifying user-interface patterns, and expressing the solution in an abstract way that is independent of a particular design or implementation is ultimately important. Only that way we can promote a language of patterns that supersedes the restrictions of a particular platform. The notational extensions provided in the Wisdom notation for both task and presentation aspects of user-interfaces enable the abstract description of user-interface patterns. Moreover, since the Wisdom notation complies with the UML standard, we can take advantage of enhanced communication with software developers, while also taking advantage of

tools support. An example of tool support for task patterns is provided in [Paternò, 2000]. Although limited to task patterns, this example illustrates how access to user-interface patterns in modeling tools can increase the efficiency of the design process by enabling designers to incorporate different UI patterns when they are producing the models. In this example a dedicated pattern manager enables designers to browse and select different task patterns to include in their models when they encounter a common problem that can be adequately solved with a given pattern. An applicability example that could take advantage of tool support is given the task model presented in Figure IV.26, which is an instantiation of the search&refine task pattern identified in [Paternò, 2000].

V.5. CONCLUSION

This chapter discussed different uses of the Wisdom method in support for user-interface design. Here we discussed the role of CASE technology, in particular user-interface tools, which support user-interface design in a way that enables designers to deal with the increasing complexity of modern user-interfaces.

The Wisdom notation is compliant with the UML standard, and permits designers to model both the task and presentation aspects of user-interfaces in conformance with the best practices of usability engineering and human-computer interaction. In this chapter we explained and demonstrated, through the description and discussion of a set of practical experiences, the capabilities of Wisdom to improve user-interface design and leverage tool support.

CASE technology is expected to increase the productivity of developers, improving the quality of the end products and, thus reduce the maintenance costs, while also making software development more enjoyable. However, several studies have pointed out that case adoption and usage is far from confirming those expectations. The UML, and the related tool interchange formats (XMI), are expected to encourage the growth of CASE usage, in particular for OO analysis and design. This is mainly because the UML enabled access to a standard non-proprietary modeling language that complies with a meta-modeling facility (MOF). Therefore vendors can focus on a single language and take advantage of flexible interoperability that permits loss-less information exchange. There is now consensus that we have access to the standards and technology that could improve the effectiveness of CASE tool usage in modern software engineering. However, software development has changed significantly in the past years, thus tools must comply with the new challenges that software developers face today. The same principles that underlie user-centered design also apply to CASE tools. Hence, the new tools must focus on the real world tasks that are ultimately important for software developers. Particularly, tools should support, not only the "hard" aspects, but also the "soft" aspects of software development. These include support for creativity, improvisation and design assistance over a process-oriented framework. This vision of CASE tools is support by market research, which indicates that the worldwide market for OO analysis and design tools should expand from adoption by smaller software development organizations.

User-interface design is irrefutably one of the most creative activities in software development. Despite that, user interface tools are one of the most successful market segments in the industry. The relative stability of the current desktop graphical user interface enabled user interface tools to reach a sophistication level that virtually any

interactive system today is built using some form of UI tool. However, the requirements for the next generation of user interfaces are quite different. The increasing diversity of computing devices, and the increasing connectivity of existing desktop computers, will have a profound effect on the future of user-interfaces. In this chapter we argue that the diversity and complexity of the modern user-interfaces imposes new needs, for enhanced notations and interoperability mechanisms, capable of maintaining the high quality tool support that we have today. The experiences described in this chapter demonstrate how Wisdom can meet those new requirements that both software developers and user interface designers will face with the advent of the next generation of interactive systems. Although the experiences were conducted at a highly experimental level, they provided results that clearly indicate that this envisioned development environment is feasible.

In section V.1.3 of this chapter we presented several experiences generating appliance~-independent user-interface description languages directly from the UML-based Wisdom notation through the XMI standard. These experiences proved that it is feasible to work with high-level descriptions of user-interfaces and produce concrete user interfaces capable of being rendered in multiple platforms. Although model-based approaches, which were popular in UI research several years ago, already proved that automatic generation was feasible, they failed to achieve commercial adoption because they didn't provide flexibility and control over the resulting UI. Moreover, model-based approaches relied on specific formalism that imposed a step learning curve for designers. The approach we described in section V.2.3, takes advantage of the forthcoming next generation appliance-independent UI description language (XForms), to achieve flexibility and control over the resulting user-interface. We propose to accomplish flexibility through separation of concerns between the abstract interaction, described in the Wisdom presentation model, and the user interface presentation specifics provided by XForms, and eventually manipulated by a graphical tool. Moreover, since our approach can take advantage of the XMI interchange format, we can exchange the models between the different tools without loosing information. This proposal also leverages artifact change because user-interface developers can use the same modeling language used by software developers. Each developer can focus on his or her task and share artifacts for increased understanding of the overall development context without requiring additional effort. Therefore we are also able to give the process context to the development tools required to increase tool adoption and usage, even in extreme environment like the ones found in small software development companies or groups.

Section V.3 takes our tool environment proposal even further by demonstrating how different tools, working over different notations, can exchange information using the XMI standard. In this section we describe another experience were the UML notational extensions for the Wisdom dialogue model can be interchanged with the ConcurTaskTrees formalism and associated tools (CTTe). Although Wisdom dialogue models can be built and manipulated in any UML modeling tool, the CTT

environment is clearly more focused and usable for the specific requirements of user interface designers. Our experiences demonstrate how we can take advantage of the XMI interchange format, to transform models between different special purpose tools (and notations), thus enabling developers to take advantage of enhanced support for their tasks. Since the interchange format permits loss-less transfer of information, the models can be interchanged back and forth, enabling consistency between models to be maintained. Moreover, automatic generation of user interfaces can be complemented with the information conveyed in dialogue models. Thus we expand our proposal for a multiple tool environment, including the information produced by the Wisdom dialogue model. Task models in this envisioned environment are used to inform the automatic generation of the user-interface specifics, thus complementing the information in the presentation model.

Section V.4 discusses how the Wisdom notation can be used to support the representation of solution for user-interface patterns. User interface patterns are one of the most promising approaches to foster reuse and deal with the increased complexity and diversity of modern user interfaces. Despite the growing interest in this area, a consensual pattern language for user-interface design as not yet emerged. In this section we argue that one of the problems preventing the dissemination of a UI patterns is the lack of a standard language to describe the recurring solutions underlying those patterns. Software patterns emerged with the advent of OO analysis and design notations, likewise we argue that UI patterns can only emerge when the community agrees on a standard language that is capable of depicting abstract solutions that are independent of a particular design or implementation. In this section we demonstrate how the Wisdom notation can be used to represent UI patterns in an abstract way. Furthermore, since the Wisdom notation is UML compliant, they share the same language commonly used to represent software patterns. Moreover, combined with the tool support described earlier, Wisdom UI patterns can be introduced as design aids in modeling tools.

VI. CONCLUSIONS AND FUTURE DEVELOPMENTS

"The most exciting phrase to hear in science, the one that heralds new discoveries, is not 'Eureka!' but 'That's funny...'"
—Isaac Asimov

This chapter describes the main conclusions of this thesis and outlines future developments.

VI.1. CONCLUSIONS

"A conclusion is the place where you got tired of thinking."
—Unknown

Building interactive systems that are efficient and reliable in use, easy to learn and remember, and that provide user satisfaction is one of the greatest challenges of the software industry. These requirements are increasing with the widespread use of the Internet, the advent of the information appliances and the increasing connectivity of existing desktop computers. In the past decades, usability engineering was consistently neglected in software development. However, we are rapidly moving away from a technology-centered foundation into a customer-centered maturity. While in the beginning computing technology could not meet all the needs of the customers, and software developers could take advantage of the relative stability of the desktop paradigm; today customers demand a diversity of highly usable and specialized information devices that are efficient, reliable and convenient.

The software industry is embracing this new reality, and small companies are leading the way. Taking advantage of the progress in software engineering, today small groups of developers are able to build complex software intensive systems in a relatively short time. However, they are faced with development processes and methods that were traditionally devised to support contract-based development by large teams of developers. Those methods and processes are not adequate for the turbulent and highly competitive markets where small companies operate, and they don't take advantage of what best characterizes them: fast reaction, enhanced communication, flexibility, improvisation and creativity. Moreover, the architectural frameworks and notations that object modeling currently offers are either complex or not adapted to support user-centered development and user-interface design.

This thesis explored how object modeling could contribute to integrate usability engineering into modern software development, providing benefits for both disciplines. The underlying principle was that some of the problems that software engineering faces today are the essence of what usability engineering promoted for years. Conversely, some of the problems that usability engineering currently faces are already successfully solved in the software engineering field.

The Wisdom UML-based approach presented here promotes a lightweight software development method encompassing three major aspects as follows:

- The Wisdom process is an alternative to the Unified Process that recognizes the needs of software engineering in the small and takes advantage of what best characterizes this development context. The Wisdom process is based on an evolutionary prototyping model that best deals with the changing requirements and critical time to market factors that SSDs usually face. This model also seamlessly rationalizes the chaotic ways of working characterizing many SSDs, thus enabling process improvement strategies. The Wisdom process controls the evolutionary prototyping model by providing an architecture-centric approach that prevents the systems from becoming badly structured, and promoting an essential use-case and task driven approach that ensures development progresses in terms of the critical added value for the customers and end-users.
- The Wisdom architecture provides two architectural models that leverage user-centered development and user-interface design. The Wisdom model architecture specifies the different models required to support user-role modeling, interaction modeling, dialogue modeling and presentation modeling. At the analysis level, Wisdom promotes a new user-interface architecture that expands the understanding of the existing UP analysis framework to include usability-engineering concerns. Therefore developers are able to integrate user-interface structural elements in the overall system architecture, fostering robustness and reuse.
- The Wisdom notation is a subset and extension of the UML that reduces the total number of concepts required to develop interactive systems. The Wisdom notation simplifies the application of the UML and defines a new set of modeling constructs necessary to support the Wisdom architectural models and the different modeling techniques used to detail those models. In particular the Wisdom notation formalizes the modeling constructs required to: model the various user roles and their relevant and salient characteristics; model system requirements in terms of essential use-cases and essential task flows; model the user-interface elements that are relevant for the overall system architecture; model the human-computer dialogue through a hierarchical task based notation; and finally model the abstract presentation of interactive system in a form independent of the underlying interaction styles and technologies.

This thesis also illustrates how usability engineering can take advantage of object modeling, and of the UML in particular, to leverage many of the new and emergent problems in user-interface design for multiple information appliances. In particular we demonstrate how the UML interchange mechanisms can: leverage automatic and flexible generation of user-interfaces from high-level models; tool interoperability promoting artifact change between UML modeling tools and task modeling tools. Finally we illustrate how abstract and standard notation can describe recurring user-

interface solutions (user-interface patterns), thus promoting the dissemination of a pattern language for interaction design and fostering reuse.

Although considerable work remains, this thesis demonstrates the feasibility of applying sound and systematic engineering principles to build interactive software of quality in turbulent and competitive environments. Today software is one of the most important industries in the world. Developing software of quality is hard and the complexity will only increase as technology progresses. Nevertheless, it is our responsibility to ensure that engineering prevails over high-speed hacking.

However, as Larry Constantine points out:

> *"Ultimately, the true pace of change is not dictated by the evolution of science or technology or of ideas, but by the capacities of humans and human social systems to accommodate change. A product, a service, a practice, or a perspective - however new and innovative - can have no impact without acceptance; no significance without change in people and their institutions".*
>
> Larry Constantine, *Back to the Future*,
> Communications of the ACM, 44(3), March 2001, p. 128.

In the end, it's all about the users, but developers are also users of the modeling languages, tools, techniques, and methods. Ensuring that they have a proper set of manageable tools, methods, and techniques is the first step toward our quest for quality and usability in software.

VI.2. REFLECTIONS AND FUTURE DEVELOPMENTS

"These days, unless you devote an enormous amount of time to anticipating the future, you won't have any future."
— Ron Chernow, business biographer

The work leading to this thesis has generated many interesting and promising ideas. Some of those future developments are promising ideas that we believe are worth exploring, others are equally interesting ideas that we have dropped during the research program due to different reasons. In the following we discuss four classes of possible extensions. Where applicable we also state some reflections about the future of object modeling for user-centered design and user-interface design.

VI.2.1. Software Process and Process Improvement

During the research program leading to this thesis it was evident that there is not enough knowledge about the actual practices carried out by software developers, in particular those working in small software companies, small teams in large software organizations or within user organizations. One initial effort was carried out to perform an extensive survey to the small software development organizations in Portugal. However, that survey never passed the test phase due to insufficient data enabling a precise characterization of the sample, and also because there was not enough interest and support from software related organizations. We still believe that such a study would be very important to provide guidelines for future research on non-classical software processes, and in particular software engineering in the small.

During the development of the research program described here the problem of benchmarking the Wisdom method emerged repeatedly. Several companies currently use the Wisdom method, and there is substantial empirical evidence that Wisdom helped improve, at least to a reasonable degree, the efficiency and control of their development processes. Despite many efforts to collect process data, those attempts failed because such systematic activities are very hard to induce in turbulent environments. Moreover, Wisdom evolved constantly during the past years and many of the adopter companies tailored the different Wisdom components to some extent. However, we still believe that a thorough empirical test should be carried out to assess the process improvement capabilities of Wisdom.

One final issue worth exploring, on the software process side of Wisdom, is process management. During this research program we have concentrated on the technical contributions underlying the method, but there is a great deal of opportunity for improvement in process management. Several companies adopting Wisdom developed their own process management activities. There is substantial empirical knowledge on those practices that is worth systematizing.

VI.2.2.UML and Notation Issues

During the different Workshops about the subject underlying this thesis, the idea that a general UML profile for user-centered development and user-interface design is needed was controversial. As the work described in this thesis reflects, much of the notational extensions provided to support user-centered development and user-interface design are highly related to their underlying development methods. To some extent the situation is similar to the "method war" that preceded the UML standardization. Moreover, there is a strong reaction from the HCI community towards some of the conventions underlying the UML. The Tupis'00 workshop, whose primary goal was to push the idea of a UML profile for UCD and UI design, completely failed that purpose (at least to date). However, we still believe that a consensus about this issue is important to foster an increasing recognition from the SE community of the improvements that usability engineering could bring to software development. Moreover, tool support, and other automation benefits, could only arise if there is consensus over a set of notation extensions that comply with the UML. There is an additional danger with failing to reach consensus over a UML profile, which is the possibility of a major tool vendor pushing a different approach that eventually becomes a *de facto* standard. A similar example already occurred with the UML profiles for data modeling and web applications that overruned significant research efforts in the fields.

One of the major contributions for the seamless adoption of Wisdom is the effectiveness of participatory sessions with end-users. The usefulness of participatory sessions highly depends on the clear mappings between the information conveyed in the sticky cards and the more formal UML extensions. While this was a primary concern in Wisdom, the need to periodically translate information between low-tech materials and UML artifacts is an important impediment to user-centered development. However, this kind of translation, given that mappings are known as they are, is clearly a feature that a tool could easily provide. Therefore, we believe that devising a new UML participatory notation, and providing tool support for that notation, could highly leverage user-centered design.

The UML extensions underlying the Wisdom notation contain several pragmatic solutions, in particular on what concerns the dialogue modeling extensions. This is because the existing UML extension mechanisms are not flexible enough for such profound extensions. The UML 2.0 RTF clearly mentions flexible extensibility as an

important requirement for the next version of the standard. If the UML 2.0 actually becomes a "language of languages", then clearly task notations are one of the most important candidates for a specific language that should be integrated into the UML. However, to support this kind of integration, not only the semantic aspects need improvement, but also flexibility at the notational level is crucial to devise usable constructs. Nevertheless, we believe that UML 2.0 will bring many new opportunities to improve the Wisdom notation.

VI.2.3.Tool issues

Much of the future developments mentioned before are closely related to CASE tool support. Process management is clearly one area that could highly benefit from adequate process-centric CASE tool support. Current UML modeling tools are not focused on the requirements of software developers. There is substantial empirical evidence that combining process (or method) information in modeling tools could highly contribute for increased CASE tool adoption. That fact is more evident in small organizations because they are highly focused on the development tools (typically 4GLs) and resist shifting tools for modeling purposes. Carefully integrating modeling features in 4GLs seams an obvious solution to this problem. In addition, conventional modeling tools do not support multiple levels of abstraction. Neglecting the fact that developers think about models at different levels of abstraction, and failing to adequately support traceability between those models, is clearly an impediment for CASE tool adoption. Moreover, modeling tools don't automate some of the more obvious tasks that developers face everyday. For instance, rules for mapping class diagrams to the relational schema are known for years, but this process is still incipient in many mainstream UML tools.

One obvious future progress of the work presented in this thesis is the development of the experiences described in Chapter 5. Automatic flexible generation of multiple user interfaces is clearly the key to the raising problem of the diversity of information appliances. We believe that the experiences described in this thesis can effectively lead to more concrete results towards this direction, in particular, when the underlying technologies mature.

INDEX

4

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

REFERENCES

Abowd, G. D. and Mynatt, E. D. [2000] Charting past, present, and future research in ubiquitous computing, *ACM Transactions on Computer-Human Interaction,* **7,** 29-58.

Abrams, M. [1999] *UIML: An Appliance-Independent XML User Interface Language,* in Proceedings of WWW8,, Toronto, Canada.

ACM [1992] *ACM SIGCHI Curricula on Human Computer Interaction,* Curricula Recommendation, Association Computing Machinery, http://www.acm.org/sigchi.

Alexander, C., Ishikawa, S. and Silverstein, M. [1977] *A pattern language : towns, buildings, construction,* Oxford University Press, New York.

Apple, C. I. [1992] *Macintosh human interface guidelines,* Addison-Wesley Pub. Co., Reading, Mass.

Artim, J. [2001] *Entity, Task and Presenter Classification in User Interface Architecture: an Approach to Organizing Practice*, In Object Modeling and User Interface Design(Ed, Harmelen, M. v.) Addison-Wesley.

Artim, J., Harmelen, M. v., Butler, K., Guliksen, J., Henderson, A., Kovacevic, S., Lu, S., Overmeyer, S., Reaux, R., Roberts, D., Tarby, J.-C. and Linden, K. V. [1998] Incorporating work, process and task analysis into industrial object-oriented systems development, *SIGCHI Bulletin,* **30**.

Atkinson, C. and Kuhne, T. [2000] *Strict Profiles: Why and How,* in Proceedings of <<UML>> 2000 - The Unified Modeling Language, Springer-Verlah, York - UK.

Azevedo, P., Merrick, R. and Roberts, D. [2000] *OVID to AUIML - User-Oriented Interface Modelling,* in Proceedings of Towards a UML Profile for Interactive System development (TUPIS'00) Workshop, http://math.uma.pt/tupis00, York - UK.

Baecker, R. M. [1992] *Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration,* Morgan Kaufmann Publishers, San Francisco, Calif.

Baecker, R. M. [1995] *Readings in human-computer interaction : toward the year 2000,* Morgan Kaufmann Publishers, San Francisco, Calif.

Barnard, P. [1987] *Cognitive resources and the learning of human-computer dialogs*, In Interfacing Thought(Ed, Carroll, J.) MIT Press, Cambridge MA, pp. 112-159.

Bass, L. [1992] A metamodel for the runtime architecture of an interactive system: The UIMS developers workshop, *SIGCHI Bulletin,* **24,** 32-37.

Baumer, D., Bischofberger, W. R., Lichter, H. and Ziillighoven, H. [1996] *User Interface Prototyping - Concepts, Tools, and Experience,* in Proceedings of International Conference on Software Engineering (ICSE'96).

Bayle, E., Bellamy, R., Casaday, G., Erickson, T., Fincher, S., Grinter, B., Gross, B., Lehder, D., Marmolin, H., Moore, B., Potts, C., Skousen, G. and Thomas, J. [1998] Towards a Pattern Language for Interaction Design: A CHI'97 Workshop, *SIGCHI Bulletin,* **30**.

Beaudouin-Lafon, M. [2000] *Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces,* in Proceedings of CHI 2000 Conference on Human Factors in Computing Systems 2000.

Beck, K. and Cunningham, W. [1989] *A laboratory for teaching object oriented thinking,* in Proceedings of OOPSLA'89.

Beck, K. and Fowler, M. [2001] *Planning extreme programming,* Addison-Wesley, Boston.

Benyon, D. and Green, T. [1995] *Displays as Data Structures,* in Proceedings of INTERACT'95, Chapman and Hall - London.

Beyer, H. and Holtzblatt, K. [1998] *Contextual Design,* Morgan Kaufmann Publishers, San Francisco, CA.

Bias, R. G. and Mayhew, D. J. [1994] *Cost-justifying usability,* Academic Press, Boston.

Blaha, M. and Premerlani, W. [1997] *Object-Oriented Modeling and Design for Database Applications,* Prentice-Hall.

Bluestone, S. I. [1999] *XwingML,* Specification, Bluestone Sogftware Inc., http://www.bluestone.com/xml/XwingML.

Bohem, B. [1988] A Spiral Model of Software Development and Enhancement, *IEEE Computer,* **21**.

Bomsdorf, B. and Szwillus, G. [1998] From Task to Dialogue: Task-Based User Interface Design, *SIGCHI Bulletin,* **30**.

Bomsdorf, B. and Szwillus, G. [1999] Tool Support for Task-based User Interface Design - A CHI'99 Workshop, *SIGCHI Bulletin,* **31**.

Booch, G. [1996] *Object solutions : managing the object-oriented project,* Addison-Wesley Pub. Co., Menlo Park, Ca.

Brodman, J. and Johnson, D. [1994] *What Small Businesses and Small Organisations say about CMM,* in Proceedings of ICSE'94, IEEE Computer Society.

Brodsky, S. [1999] *XMI Opens Application Interchange,* White Paper, IBM Corporation,

Browne, D. [1993] *STUDIO: STructured User-interface Design for Interaction Optimisation,* Prentice Hall.

Button, G. and Dourish, P. [1996] *Technometodology: Paradoxes and Possibilities,* in Proceedings of CHI'96, ACM Press.

Card, S. K., Moran, T. P. and Newell, A. [1983] *The psychology of human-computer interaction,* L. Erlbaum Associates, Hillsdale, N.J.

Castellani, X. [1999] *Overviews of Models Defined with Charts of Concepts,* in Proceedings of Information System Concepts: An Integrated Discipline Emerging, IFIP WG 8.1 International Working Conference (ISCO4), Kluwer Academic Publishers.

Cattaneo, F., Fuggetta, A. and Lavazza, L. [1995] *An experience in process assessment,* in Proceedings of ICSE'95.

Cerf, V. [1992] *A perspective on ubiquitous computing,* in Proceedings of ACM annual conference on Communications,, Kansas City, MO USA.

Choust, G., Grünbacher, P. and Schoisch, E. [1997] *To Spire or not to Spire: that is the Question,* in Proceedings of EUROMICRO'97 Short Contributions.

Cockburn, A. [1997] Structuring Use Cases with Goals, *Journal of Object-Oriented Programming*.

Cockburn, A. [2000] *Writing Effective Use Cases,* Addison-Wesley.

Collins, D. [1995] *Designing object-oriented user interfaces,* Benjamin Cummings, Redwood City, CA.

Conallen, J. [1999] *Building Web Applications with UML,* Addison-Wesley.

Constantine, L. [1992] Essential Modeling: Use cases for user interfaces, *ACM Interactions*.

Constantine, L. L. [2000] *The Usability Challenge: Can UML and the Unified Process Meet It?,* in Proceedings of TOOLS Pacific'2000,, Sidney.

Constantine, L. L. and Lockwood, L. A. D. [1999] *Software for use : a practical guide to the models and methods of usage-centered design,* Addison Wesley, Reading, Mass.

Constantine, L. L. and Lockwood, L. [2001] *Structure and Style in Use Cases for User Interface Design,* In Object Modeling and User Interface Design(Ed, Harmelen, M. v.) Addison-Wesley.

Cook, S. [2000] *The UML Family,* in Proceedings of <<UML>> 2000 - The Unified Modeling Language, Springer-Verlag, York, UK.

Cook, S., Kleppe, A., Mitchell, R., Rumpe, B., Warmer, J. and Wills, A. [1999] *Defining <<UML>> Family Members Using Prefaces,* in Proceedings of TOOLS Pacific, IEEE Computer Society.

Cooper, A. [1999] *The inmates are running the asylum,* Sams, Indianapolis, IN.

Coutaz, J. [1987] *PAC: An object-oriented model for dialogue design,* in Proceedings of INTERACT'87, Elsevier Science Publisher.

Coutaz, J. [1993] *Software Architecture Modeling for User Interfaces*, In Encyclopedia of Software EngineeringWiley.

Daly-Jones, O., Bevan, N. and Thomas, C. [1999] *Handbook of User-centered Design,* Serco Usability Services,

Dayton, T., McFarland, A. and Kramer, J. [1998] *Bridging User Needs to Object Oriented GUI Prototype via Task Object Design*, In User Interface Design(Ed, Wood, L. E.) CRC Press, Boca Raton - Florida - EUA, pp. 15-56.

Dix, A. [1999] *Designing of User Interfaces for the Web,* in Proceedings of User Interfaces for Data Intensive Systems (UIDIS'99),, Edimburgh, Scotland.

Dix, A. J. [1998] *Human-computer interaction,* Prentice Hall Europe, London ; New York.

DSouza, D. F. and Wills, A. C. [1999] *Objects, components, and frameworks with UML : the catalysis approach,* Addison-Wesley, Reading, Mass.

Duce, D., Hopgood, D. and Gomes, M. R. (Eds.) [1991] *User Interface Management and Design,* Springer Verlag.

Dyba, T. [2000] Improvisation in Small Software Organizations, *IEEE Software,* **17,** 82-87.

EITO [1999] *European Information Technology Observatory,* European Economic Interest Grouping (EEIG).

Finkelstein, A. and Kramer, J. [2000] *Software Engineering: A Roadmap*, In The Future of Software Engineering(Ed, Finkelstein, A.) ACM, New-York.

Foley, J., Kim, W. C., Kovacevic, S. and Murray, K. [1991] *UIDE — An Intelligent User Interface Design Environment*, In Architectures for Intelligent Interfaces: Elements and Prototypes(Eds, Sullivan, J. and Tyler, S.) Addison-Wesley.

Foley, J. D., Dam, A. v., Feiner, S. K. and Hughes, J. F. [1990] *Computer Graphics: Principles and Practice,* Addison-Wesley, Reading - Mass.

Fowler, M. [1996] *Analysis Patterns: Reusable Object Models,* Addison-Wesley, Reading, Mass.

Fowler, M. and Scott, K. [1999] *UML distilled : a brief guide to the standard object modeling language,* Addison Wesley, Reading, Mass.

Fuggetta, A. [2000] *Software Process: A Roadmap*, In The Future of Software Engineering(Ed, Finkelstein, A.) ACM, New York.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. [1995] *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley Longman, Reading Mass.

Golberg, M. and Robson, D. [1983] *Smalltalk-80: The language and its implementation,* Addison-Wesley, Reading, MA.

Gould, J. D. and Lewis, C. [1985] Designing for Usability: Key principles and what designers think, *Communications of the ACM,* **28,** 300-311.

Graham, I. [1994] *Object-oriented methods,* Addison-Wesley, Wokingham, England ; Reading, Mass.

Graham, I. [1995] *Migrating to object technology,* Addison-Wesley Pub. Co., Wokingham, England ; Reading, Mass.

Graham, I. [1998] *Requirements engineering and rapid development : an object-oriented approach,* Addison Wesley, Harlow, England ; Reading, MA.

Green, T. [1991] *Describing Information Artifacts with Cognitive Dimensions and Structure Maps*, In People and Computer V(Eds, Diaper, D. and Hammond, N. V.) Cambridge University Press, Cambridge.

Green, T. and Benyon, D. [1996] The Skull Beneath the Skin: Entity Relationships Modeling of Information Artefacts, *International Journal of Human Computer Studies,* **44,** 801-828.

Grudin, J. [1991] Interactive Systems: Bridging the Gap between Developers and Users, *IEEE Computer,* **24,** 59-69.

Gulliksen, J., Göransson, B. and Lif, M. [2001] *A user centered design approach to object oriented user interface design*, In Object Modeling and User Interface Design(Ed, Harmelen, M. v.) Addison-Wesley.

Hackos, J. T. and Redish, J. C. [1998] *User and Task Analysis for Interface Design,* John Wiley & Sons.

Harmelen, M. v. [1996] *Object-Oriented Modelling and Specification for User Interface Design,* in Proceedings of Workshop on Design, Specification and Verification of Interactive

Systems (DSV-IS'96), Springer-Verlag.

Harmelen, M. v. (Ed.) [2001a] *Object Modeling and User Interface Design,* Addison-Wesley.

Harmelen, M. v. [2001b] *Designing with Idiom*, In Object Modeling and User Interface Design(Ed, Harmelen, M. v.) Addison-Wesley.

Harmelen, M. v. [2001c] *Interactive System Design using OO&HCI Methods*, In Object Modeling and User Interface Design(Ed, Harmelen, M. v.) Addison-Wesley.

Harmelen, M. v., Artim, J., Butler, K., Henderson, A., Roberts, D., Rosson, M.-B., Tarby, J.-C. and Wilson, S. [1997] Object Models in User Interface Design, *SIGCHI Bulletin,* **29**.

Hill, R. D. [1993] *The Rendezvous Constraint Maitenence System,* in Proceedings of ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST'93), ACM, New York, Atlanta, Ga.

Hill, R. D. and Herrmann, M. [1987] *The Structure of Tube-A tool for Implementing Advanced User Interfaces,* in Proceedings of Eurographics'89, Elsevier, North-Holland.

Hix, D. and Hartson, H. R. [1993] *Developing user interfaces : ensuring usability through product & process,* J. Wiley, New York.

Hollan, J., Hutchins, E. and Kirsh, D. [2000] Distributed Cognition: Toward a New

Foundation for Human-Computer

Interaction Research, *ACM Transactions on Computer-Human Interaction,* **7,** 174-196.

Hudson, W. [2001] *Towards Unified Models in User-Centered and Object-Oriented Design*, In Object Modeling and User Interface Design(Ed, Harmelen, M. v.) Addison-Wesley.

Hurlbut, R. R. [1998] *Managing Domain Architecture Evolution Through Adaptive Use Case and Business Rule Models,* DSc, Illinois Institute of Technology.

IBM [2000a] *Visual Age for Java,* IBM Corporation, http://www.ibm.com.

IBM [2000b] *Visual Age Team Connection,* IBM Corporation, http://www.ibm.com.

IBM, C. [1991] *Common User Access Guide to User Interface Design: Systems Application Architecture,* IBM.

IBM, C. [1999] *DRUID - A Language for Marking-up Intent-based User Interfaces Version 1.1,* Internal Technical Report (Confidential), IBM Corporation,

IIvari, J. [1996] Why are CASE Tools Not Used?, *Communications of the ACM,* **39,** 94-103.

Isensee, S. and Rudd, J. [1996] *The art of rapid prototyping : user interface design for Windows and OS/2,* International Thomson Computer Press, London ; Boston.

ISO [1998] *ISO9241 - Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs) - Part 11 Guidance on Usability,* International Organisation for Standardisation.

ISO [1999] *ISO 13407 Human Centred Design Process for Interactive Systems,* International Organisation for Standardisation.

Jacobson, I. [1992] *Object-oriented software engineering: a use case driven approach,* ACM Press - Addison-Wesley Pub., New York.

Jacobson, I., Ericsson, M. and Jacobson, A. [1995] *The object advantage : business process reengineering with object technology,* Addison-Wesley, Wokingham, England ; Reading, Mass.

Jacobson, I., Booch, G. and Rumbaugh, J. [1999] *The unified software development process,* Addison-Wesley, Reading, Mass.

Jarzabek, S. and Huang, R. [1998] The Case for User-Centered CASE Tools, *Communications of the ACM,* **41,** 93-99.

JOCE [1996] *European Comission Recommendation N. 96/280/CE,* European Comission,

John, B. E. [1990] *Extensions of GOMS analyses to expert performance requiring perception of dynamic visual and auditory information,* in Proceedings of Human Factors in Computer Systems (CHI), ACM.

John, B. E. [1995] Why GOMS?, *ACM Interactions,* **,** 80-89.

John, B. E. and Gray, W. D. [1995] *CPM-GOMS: An analysis method for tasks with parallel activities,* in Proceedings of Human Factors in Computing Systems (CHI '1995), ACM.

John, B. E. and Kieras, D. E. [1996a] Using GOMS for User Interface Design and Evaluation: Which Technique?, *ACM Transactions on Computer-Human Interaction,* **3,** 287-319.

John, B. E. and Kieras, D. E. [1996b] The GOMS Family of User Interface Analysis Techniques: Comparison and Contrast, *ACM Transactions on Computer-Human Interaction,* **3,** 320-351.

Johnson, P., Johnson, H. and Wilson, S. [1995] *Rapid Prototyping of User Interfaces Driven by Task Models*, In Scenario-Based Design – Envisioning Work and Technology in System Development(Ed, Carroll, J. M.) John Wiley & Sons.

Jones, C. B. [1986] *Systematic Software Development Using VDM,* Prentice-Hall.

Karat, C.-M. [1990] *Cost-benefit Analysis of Usability Engineering Techniques,* in Proceedings of Human Factor Society 34th Annual Meeting, Human Factors Society.

Karat, J. [1997] Evolving the Scope of User-Centered Design, *Communications of the ACM,* **40,** 33-38.

Kemerer, C. F. [1992] How the Learning Curve Affects CASE Tool Adoption, *IEEE Software,* **9,** 23-28.

Khare, R. [2000a] How <FORM> Functions: Transcending the Web as GUI, Part I, *IEEE Internet Computing*,**,** 88-90.

Khare, R. [2000b] Can XForm Transform the Web? Transcending the Web as GUI - Part II, *IEEE Internet Computing*,**,** 103-106.

Kieras, D. E. [1988] *Towards a practical GOMS model methodology for user interface design*, In The Handbook of Human-Computer InteractionAmsterdam, pp. 135–158.

Kieras, D. E., Wood, S. D. and Meyer, D. E. [1997] Predictive Engineering Models Based on the EPIC Architecture for a Multimodal High-Performance Human-Computer Interaction Task, *ACM Transactions on Computer-Human Interaction,* **4,** 230-275.

Kobryn, C. [1999] UML 2001: a standardization odyssey, *Communications of the ACM,* **42,** 29-37.

Kovacevic, S. [1994] *TACTICS – A Model-Based Framework for Multimodal Interaction,* in Proceedings of AAAI Spring Symposium on Intelligent Multi-Media Multi-Modal Systems.

Kovacevic, S. [1998] *UML and User Interface Design,* in Proceedings of UML'98,, Mulhouse - France.

Kovacevic, S. [1999] *Beyond Automatic Generation - Exploratory Approach to UI Design,* in Proceedings of 3rd International Conference on Computer-Aided Design of User Interfaces (CADUI'99), Kluwer, Louvain-la-Neuve, Belgium.

Kreitzberg, C. [1996] *Managing for Usability*, In Multimedia: A management perspective(Ed, Antone, F.) Wadsworth.

Kruchten, P. [1995] The "4+1" View Model of Software Architecture, *IEEE Software,* **12**.

Kruchten, P. [1998] *The Rational Unified Process: an Introduction,* Addison-Wesley.

Laitinen, M., Fayad, M. and Ward, R. [2000] Software Engineering in the Small, *IEEE Software,* **17,** 75-77.

Landauer, T. [1997] *The Trouble with Computers,* MIT Press, Cambridge, Mass.

Laudon, K. C. and Laudon, J. P. [2000] *Management information systems : organization and technology in the networked enterprise,* Prentice Hall, Upper Saddle River, NJ.

Laurel, B. and Mountford, S. J. [1990] *The Art of human-computer interface design,* Addison-Wesley Pub. Co., Reading, Mass.

Lif, M. [1999] User interface Modelling — adding usability to use cases, *International Journal of Human-Computer Studies*, **3**.

Lonczewski, F. and Schreiber, S. [1996] *The FUSE-System: an Integrated User Interface Design Environment,* in Proceedings of 3rd International Conference on Computer-Aided Design of User Interfaces (CADUI'99), Kluwer, Louvain-la-Neuve, Belgium.

Lu, S., Paris, C. and Linde, K. V. [1998] *Towards Automatic Generation of Task Models from Object-Oriented Diagrams,* in Proceedings of Working Conference on Engineering Human-Computer Interaction (EHCI'98),, Crete, Greece.

Lu, S., Paris, C. and Linden, K. V. [1999] *Towards the automatic generation of task models from object oriented diagrams,* in Proceedings of Engineering for Human-Computer Interaction (EHCI'99), Kluwer academic publishers.

March, J. G. [1991] Exploration and Exploitation in Organizational Learning, *Organizational Science,* **2,** 71-87.

Markopoulos, P. and Marijnissen, P. [2000] *UML as a representation for Interaction Design,* in Proceedings of OZCHI.

Martin, J. and Odell, J. J. [1998] *Object-oriented methods : a foundation,* Prentice Hall PTR, Upper Saddle River, N.J.

Mayhew, D. J. [1999] *The usability engineering lifecycle : a practioner's handbook for user interface design,* Morgan Kaufmann Publishers, San Francisco, Calif.

McMenamin, S. M. and Palmer, J. F. [1984] *Essential systems analysis,* Yourdon Press, New York, N.Y.

Merrick, R. [1999] *DRUID A Language for Marking-up Intent-based User Interfaces,* Internal Draft Specification, IBM Corporation,

Microsoft [2000] *Microsoft Component Technology*, Microsoft Corporation, http://www.microsoft.com/com.

Microsoft, P. [1995] *The Windows interface guidelines for software design*, Microsoft Press, Redmond, WA.

Motorola [2001] *Ther XML Cover Pages: VoxML*, Oasis Open Org., http://www.oasis-open.org/cover/voxML.html.

Mozilla [1999] *XUML Language Specification*, Specification, Mozilla, http://www.mozilla.org/xpfe.

Muller, M., Tudor, L. G., Wildman, D. M., White, E. A., Root, R. W., Dayton, T., Carr, R., Diekmann, B. and Dykstra-Erickson, E. [1995] *Bifocal Tools for Scenarios and Representations in Participatory Activities with Users*, In Scenario-Based Design: Envisioning Work and Technology in System Development(Ed, Carroll, J. M.) John Wiley & Sons.

Muller, M. J., Wildman, D. M. and White, E. A. [1993] Taxonomy of PD Practices: A Brief Practitioner's Guide, *Communications of the ACM*, **36**.

Myers, B. [1994] Challenges of HCI Design and Implementation, *ACM Interactions*,, 73-83.

Myers, B. [1995] User Interface Software Tools, *ACM Transactions on Computer-Human Interaction*, **2,** 64-103.

Myers, B. [1996] User Interface Software Technology, *ACM Computing Surveys*, **28,** 189-191.

Myers, B. and Rosson, M. B. [1992] *Survey on User Interface Programming*, in Proceedings of CHI92 Conference, ACM.

Myers, B., Hudson, S. and Pausch, R. [2000] Past, Present, and Future of User Interface Software Tools, *ACM Transactions on Computer-Human Interaction*, **7,** 3-28.

Myers, B., Giuse, D. A., Dannenberg, R. B., Zanden, B. V., Kosbie, D. S., Pervin, E., Mckish, A. and Marchal, P. [1990] Garnet: A Comprehensive Support for Graphical, Highly-Interactive User Interfaces, *IEEE Computer*, **23,** 71-85.

Nielsen, J. [1993] *Usability engineering*, Academic Press, Boston.

Nielsen, J. [1994] *Enhancing the Explanatory Power of Usability Heuristics*, in Proceedings of CHI'94, ACM.

Nielsen, J. [2000] *Designing Web Usability*, New Riders Publishing, Indianapolis, Ind.

Norman, D. A. [1988] *The psychology of everyday things*, Basic Books, New York.

Norman, D. A. [1993] *Things that make us smart : defending human attributes in the age of the machine*, Addison-Wesley Pub. Co., Reading, Mass.

Norman, D. A. [1998] *The invisible computer : why good products can fail, the personal computer is so complex, and information appliances are the solution*, MIT Press, Cambridge, Mass.

Norman, D. A. and Draper, S. W. [1986] *User centered system design : new perspectives on human-computer interaction*, L. Erlbaum Associates, Hillsdale, N.J.

Nunes, N. J. [1997] *Validação de Metodologias e Técnicas de Engenharia de Software na Concepção na Implementação de um Sistema de Supervisão para Centrais de Produção de Energia Eléctrica*, MPhil, Universidade da Madeira.

Nunes, N. J. [1999] *A Bridge too far: Can UML help bridge the gap?*, in Proceedings of IOS Press, Edimburgh - Scotland.

Nunes, N. J. and Cunha, J. F. e. [1998] *Case Study: SITINA – A Software Engineering Project Using Evolutionary Prototyping*, in Proceedings of CAiSE'98/IFIP 8.1 EMMSAD'98 Workshop,, Pisa - Italy.

Nunes, N. J. and Cunha, J. F. e. [1999] *Detailing use-case with activity diagrams and object-views*, in Proceedings of Workshop on Integrating Human Factors into Use Cases and OO Methods, http://www.crim.ca/~aseffah/ecoop/, Lisbon - Portugal.

Nunes, N. J. and Cunha, J. F. e. [2000a] *Wisdom – A UML based architecture for interactive systems*, in Proceedings of DSV-IS'2000, Springer-Verlag, Limerick - Ireland.

Nunes, N. J. and Cunha, J. F. e. [2000b] *Towards a UML profile for interaction design: the Wisdom approach*, in Proceedings of UML'2000, Springer-Verlag, York - UK.

Nunes, N. J. and Cunha, J. F. e. [2000c] Wisdom: A Software Engineering Method for Small Software Development Companies, *IEEE Software*, **17,** 113-119.

Nunes, N. J. and Cunha, J. F. e. [2001a] *Whitewater Interactive System Development with Object Models*, In Object Modeling and User Interface Design(Ed, Harmelen, M. v.) Addison-Wesley.

Nunes, N. J. and Cunha, J. F. e. [2001b] *Designing Usable Software Products: The Wisdom UML-based Lightweight Method,* in Proceedings of International Conference on Enterprise Information Systems (ICEIS'2001),, Setúbal, Portugal.

Nunes, N. J., Cunha, J. F. e. and Castro, J. [1998] *Prototipificação Evolutiva Centrada nos Utilizadores: Um Estudo de Case para Melhorar o Processo de Desenvolvimento de Software em PMEs,* in Proceedings of CLEI'98 Conference, Pontificia Universidade Catolica del Ecuador, Quito Ecuador.

Nunes, N. J., Artim, J., Cunha, J. F. e., Kovacevic, S., Roberts, D. and Harmelen, M. v. [2000] *Tupis'2000 Workshop - Towards a UML Profile for Interactive System development,* in Proceedings of <<UML>> 2000 International Conference on the Unified Modeling Languge,, York - UK.

Nunes, N. J., Toranzo, M., Cunha, J. F. e., Castro, J., Kovacevic, S., Roberts, D., Tarby, J.-C., Collins-Cope, M. and Harmelen, M. v. [1999] *Interactive System Design with Object Models (WISDOM'99),* In ECOOP'99 Workshop Reader, Vol. 1743 (Eds, Moreira, A. and Demeyer, S.) Springer-Verlag,, pp. 267-287.

OMG [1999] *Unified Modeling Language 1.3,* Standard, Object Management Group, http://www.omg.org.

OMG [2000] *OMG XML Metadata Interchange (XMI) Specification V.1.1,* Specification, Object Management Group,

OMG [2001] *The Common Object Request Broker Architecture (CORBA) Specification 2.4.2,* Specification, Object Management Group,

Paternò, F. [2000] *Model Based Design and Evaluation of Interactive Applications,* Springer-Verlag, London.

Paternò, F. [2001] *Towards a UML for Interactive Systems,* in Proceedings of International Conference on Engineering Human Computer Interaction (EHCI'2001),, Canada.

Paternò, F., Mancini, C. and Lenzi, R. [2000] *Specification of ConcurTaskTrees Task Models in XML,* Guitarre Project Technical Report, CNUCE - C.N.R.,

Paulk, M. C. [1995] *The capability maturity model : guidelines for improving the software process,* Addison-Wesley Pub. Co., Reading, Mass.

Perzel, K. and Kane, D. [1999] *Usability Patterns for Applications on the World Wide Web,* in Proceedings of PLoP'1999.

Pfaff, G. and Haguen, P. J. W. t. (Eds.) [1985] *User Interface Management Systems,* Springer-Verlag, Berlin.

Philips, C. [1995] LeanCuisine+: An Executable Graphical Notation for Describing Direct Manipulation Interfaces, *Interacting with Computers,* **7,** 49-71.

Philips, C. and Scogings, C. [2000] *Task and Dialogue Modeling: Bridging the Divide with LeanCuisine+,* in Proceedings of First Australasian User Interface Conference,, Canberra, Australia.

Preece, J. [1995] *Human-computer interaction,* Addison-Wesley Pub. Co., Wokingham, England ; Reading, Mass.

Puerta, A. [1993] *The Study of Models of Intelligent Interfaces,* in Proceedings of Workshop Intelligent User Interfaces, ACM Press, New York.

Puerta, A. [1996] *The MECANO Project: Comprehensive and Integrated Support for Model-Based Interface Development,* in Proceedings of 3rd International Conference on Computer-Aided Design of User Interfaces (CADUI'99), Kluwer, Louvain-la-Neuve, Belgium.

Puerta, A. [1997] A Model Based Interface Development Environment, *IEEE Software,*, 40-47.

Puerta, A. [1998] *Reflections on Model-Based Automated Generation of User Interfaces*, In Readings in Intelligent User Interfaces(Ed, Maybury, M. T.) Morgan-Kaufmann.

Raskin, J. [2000] *The humane interface : new directions for designing interactive systems,* Addison Wesley, Reading, Mass.

Rational [2000] *Rational Rose,* Internet site, Rational Software Corporation, http://www.rational.com.

Redmond-Pyle, D. and Moore, A. [1995] *Graphical User Interface Design and Evaluation,* Prentice-Hall, London.

Rijken, D. [1994] The Timeless Way... the Design of Meaning, *SIGCHI Bulletin,* **6**.

Robbins, J. [1999] *Cognitive Support Features for Software Development Tools,* PhD Thesis, University of California Irvine.

Roberts, D., Berry, D., Isensee, S. and Mullaly, J. [1998] *Designing for the user with OVID : bridging user interface design and software engineering,* Macmillan Technical Pub., Indianapolis, IN.

Royce, W. [1998] *Software project management : a unified framework,* Addison-Wesley, Reading, Mass.

Rumbaugh, J., Jacobson, I. and Booch, G. [1999] *The unified modeling language reference manual,* Addison-Wesley, Reading, Mass.

Rumbaugh, J., Premerlani, W., Eddy, F. and Lorensen, W. [1991] *Object-Oriented Modeling and Design,* Prentice-Hall.

Schmucker, K. J. [1986] MacApp: An Application Framework, *Byte,* **11,** 189-193.

Sears, A. [1995] *AIDE: A Step Toward Metric-Based Interface Development Tools,* in Proceedings of ACM Symposium on User Interface Software and Technology (UIST'95).

Sharma, S. and Rai, A. [2000] CASE Deploymentg in IS Organizations, *Communications of the ACM,* **41,** 80-88.

Shaw, M. and Garlan, D. [1996] *Software Architecture,* Prentice-Hall, New Jersey.

Shneiderman, B. [1982] The Future of Interactive Systems and the Emergence of Direct Manipulation, *Behaviour and Information Technology,* **1,** 237-256.

Shneiderman, B. [1998] *Designing the user interface : strategies for effective human-computer-interaction,* Addison Wesley Longman, Reading, Mass.

Silva, P. P. d. and Paton, N. W. [2000] *<<UMLi>>: The Unified Modeling Language for Interactive Applications,* in Proceedings of <<UML>> 2000 The Unified Modeling Language, Springer-Verlag, York-Kent.

Sommerville, I. [1996] *Software engineering,* Addison-Wesley Pub. Co., Wokingham, England ; Reading, Mass.

Stary, C., Vidakis, N., Mohacsi, S. and Nagelholz, M. [1997] *Workflow-oriented Prototyping for the Development of Interactive Software,* in Proceedings of 21st International Computer Software and Application Conference (COMPSAC'97).

Sukaviriya, P. and Foley, J. [1990] *Coupling a UI Framework with Automatic Generation of Context-Sensitive Animated Help,* in Proceedings of ACM Symp. on User Interface Software and Technology (UIST'90), ACM.

Sun [1997] *Java Speech Markup Language Specification V. 0.5,* Specification, Sun Microsystems, http://www.sun.com.

Sun [2000] *JavaBeans Component Architecture for Java,* Sun Microsystems, http://java.sun.com/products/javabeans.

Sutcliffe, A. and Benyon, D. [1998] *Domain modelling for interactive systems design,* Kluwer Academic Publishers, Boston.

Symantec [2000] *Symantec Café,* Symantec Corporation, http://www.symantec.com.

Szekely, P. [1995] *Declarative Interface Models for User Interface Construction Tools: The Mastermind Approach,* in Proceedings of Engineering for Human-Computer Interaction (EHCI'95), Chapman & Hall, London.

Szekely, P., Luo, P. and Neches, R. [1992] *Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design,* in Proceedings of CHI'92, ACM.

Szekely, P., Sukaviriya, P., Castells, P., Muthukumarasamy, J. and Salcher, E. [1996] *Declarative interface models for user interface construction tools: the Mastermind approach,* in Proceedings of Engineering for Humand-Computer Interaction 1996, Chapman & Hall.

Tarby, J.-C. and Barthet, M. F. [1996] *The Diane+ Method,* in Proceedings of Computer-Aided Design of User Interfaces (CADUI'96), Presses Universitaires de Namur, Namur, Namur - Belgium.

Tidwell, J. [1998a] *Interaction Design Patterns,* in Proceedings of PLoP'98.

Tidwell, J. [1998b] *Common Ground: A Pattern Language for Human-Computer Interface Report,* MIT, http://www.mit.edu/~ejtidwell/.

TogetherSoft [2000] *TogetherJ,* TogetherSoft, http://www.togethersoft.com/.

Took, R. [1990] *Surface Interaction: A Paradigm and Model for Separating Application and Interface,* in Proceedings of Human Factors in Compuring Systems (CHI'90), ACM Press.

UIML [2000] *User Interface Markup Language Draft 2.0,* Draft Specification, UIML.org, http://www.uiml.org.

Unisys [2000] *Unisys Universal Repository,* Unisys Corporation, http://www.unisys.com.

USCensus, B. [1997] *1992 Enterprise Statistics: Company Summary,* U.S. Department of Commerce, Economics and Statistics Administration, Bureau of the Census,

USCensus, B. [1999] *Advance 1997 Economic Census - Core Business Statistics Series,* U.S. Department of Commerce, Economics and Statistics Administration, Bureau of the Census.

Vanderdonckt, J. and Bodart, F. [1993] *Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection,* in Proceedings of Human Factors in Computing Systems (InterCHI '93), ACM Press, New York.

W3C [1998a] *Cascading Style Sheets, level 2 - CSS2 Specification,* W3C Recommendation, World Wide Web Consortium,

W3C [1998b] *Document Object Model (DOM) Level 1 Specification - Version 1.0,* W3C Recommendation, World Wide Web Consortium,

W3C [1999a] *HTML 4.01 Specification,* W3C Recommendation, World Wide Web Consortium, http://www.w3.org/TR/html4/.

W3C [1999b] *XSL Transformations (XSLT) - Version 1.0,* W3C Recommendation,, http://www.w3.org/TR/xslt.

W3C [2000a] *Extensible Markup Language (XML) 1.0 (Second Edition),* W3C Recommendation, W3 Consortium, http://www.w3.org.

W3C [2000b] *XHTML™ 1.0: The Extensible HyperText Markup Language - A Reformulation of HTML 4 in XML 1.0,* W3C Recommendation, World Wide Web Consortium, http://www.w3.org/TR/xhtml1/.

W3C [2000c] *Extensible Stylesheet Language (XSL) - Version 1.0,* W3C Candidate Recommendation,, http://www.w3.org/TR/xsl/.

W3C [2001] *XForms 1.0,* W3C - Working Draft, W3 Consortium, http://www.w3.org.

Wand, Y., Storey, V. C. and Weber, R. [2000] An Ontological Analysis of the Relationship Construct in Conceptual Modeling, *ACM Transactions on Database Systems,* **24,** 495-528.

Want, R. and Borriello, G. [2000] Survey on Information Appliances, *IEEE Computer Graphics & Applications,* **20,** 24-31.

WAPF [1999] *Official Wireless Application Protocol,* John Wiley and Sons.

Weiser, M. [1993] Some computer science issues in ubiquitous computing, *Communications of the ACM,* **36,** 75-84.

Welie, M. v. and Troedtteberg, H. [2000] *Interaction Patterns in User Interface,* in Proceedings of PLoP 2000.

Wiecha, C. [1990] ITS: A Tool for Rapidly Developing Interactive Applications, *ACM Transactions on Information Systems,***,** 204-236.

Wilson, S. and Johnson, P. [1996] *Bridging the Generation Gap: From Work Tasks to User Interface Designs,* in Proceedings of 3rd International Conference on Computer-Aided Design of User Interfaces (CADUI'99), Kluwer, Louvain-la-Neuve, Belgium.

Windriver [2000] *Windriver Sniff+,* Wind River System Inc., http://www.windriver.com.

Wirfs-Brock, R. [1993] Designing Scenarios: Making the Case for a Use Case Framework, *Smalltalk Report,* **Nov.-Dec.**

Yourdon, E. and Constantine, L. L. [1979] *Structured design : fundamentals of a discipline of computer program and systems design,* Prentice Hall, Englewood Cliffs, N.J.