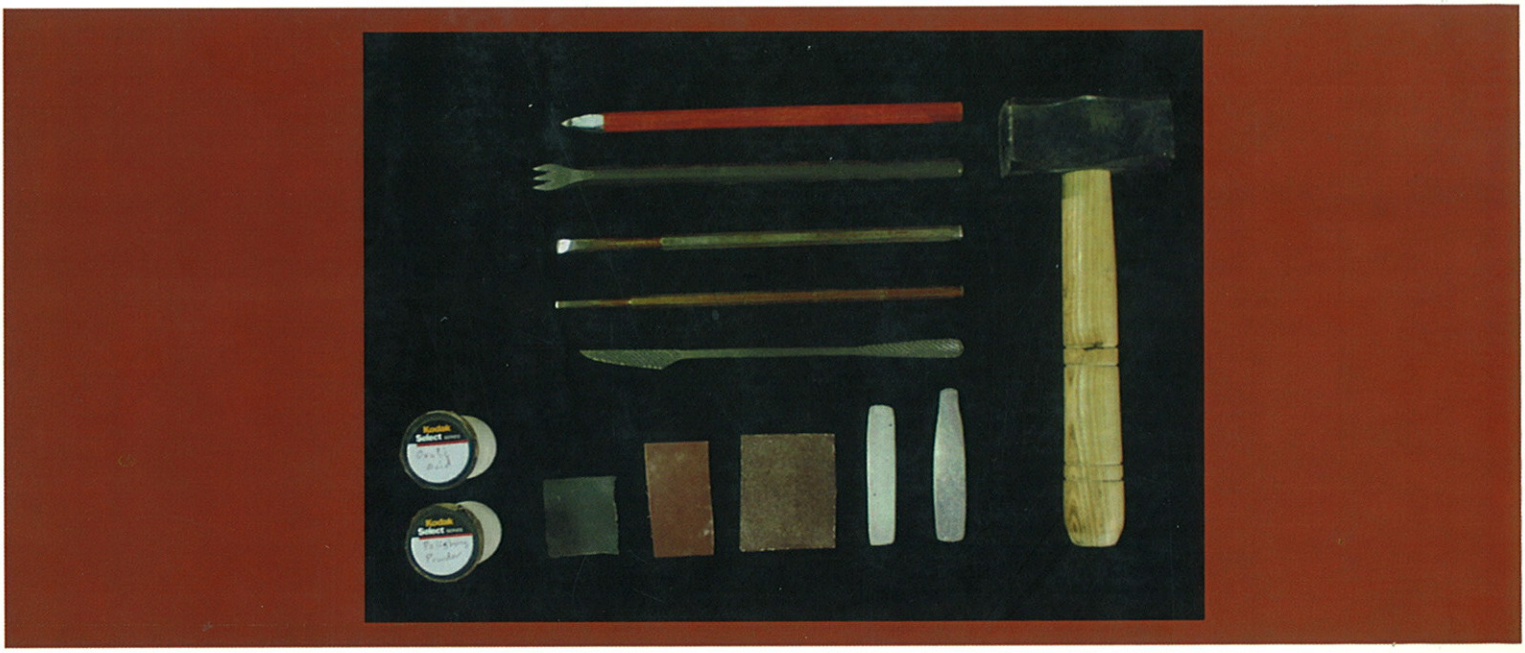
UNIVERSIDADE da MADEIRA

DESIGNING FOR WORKSTYLE TRANSITIONS: Interaction Design Tools for Software Engineering



PEDRO FILIPE PEREIRA CAMPOS
(MESTRE)

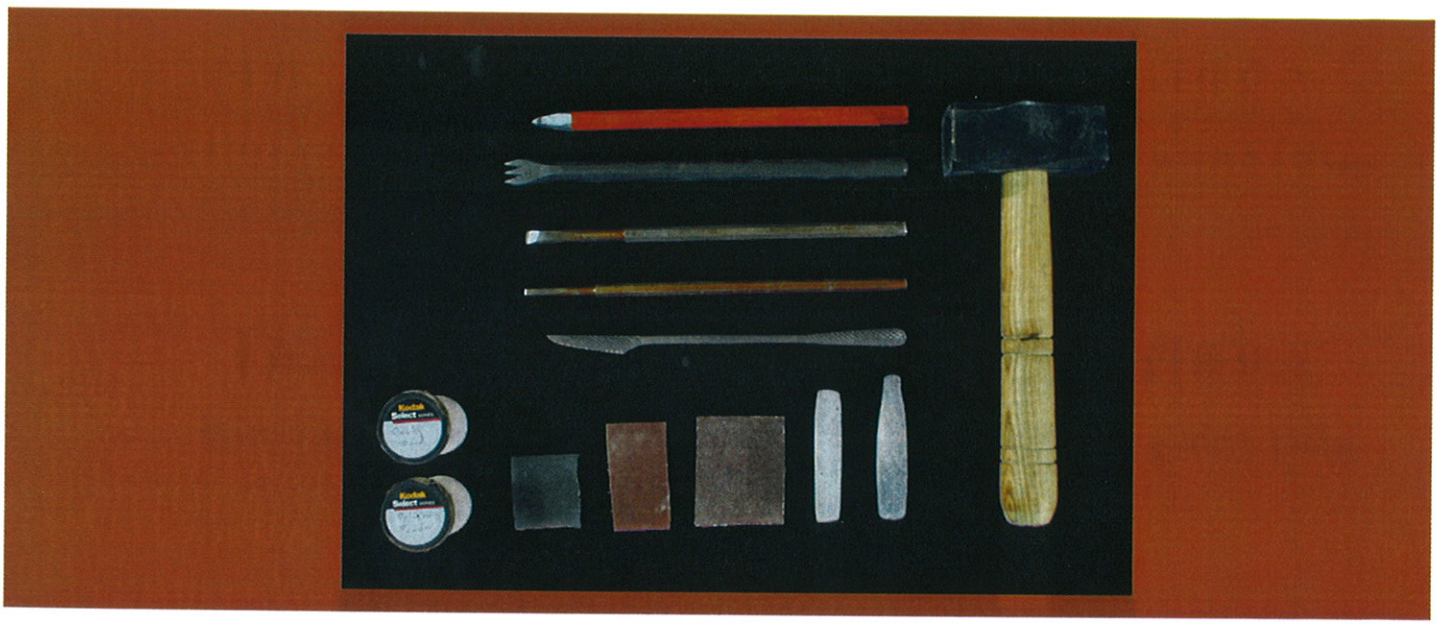
Tese submetida à Universidade da Madeira para a Obtenção do Grau de Doutor em Engenharia Informática, especialidade de Interacção Homem-Máquina

Funchal – Portugal

June 2006

UNIVERSIDADE da MADEIRA

DESIGNING FOR WORKSTYLE TRANSITIONS: Interaction Design Tools for Software Engineering



PEDRO FILIPE PEREIRA CAMPOS
(MESTRE)

Tese submetida à Universidade da Madeira para a Obtenção do Grau de Doutor em Engenharia Informática, especialidade de Interacção Homem-Máquina

Funchal – Portugal

June 2006

Designing for Workstyle Transitions: Interaction Design Tools for Software Engineering

Cover Design by Pedro F. Campos
Cover Illustration: Set of Carving Tools, Photograph by Marc Levoy, 1999.

Supervisor:

Professor Doutor Duarte Nuno Jardim Nunes

Professor Associado do Departamento de Matemática e Engenharias da Universidade da Madeira

Abstract

Software engineering (SE) models and methods, as well as the tools supporting them, have largely ignored User Interface (UI) design issues. And the lack of usability present in many of the currently used design tools has lead both interaction designers and software engineers to a low level of satisfaction with their current tools of the trade.

A new workstyle model for the User-Centered Design (UCD) process is described. The proposal is new because it is the first workstyle model tailored to UCD. The workstyle model was based on the identification of the main obstacles to UCD and SE integration, current research results, extensive observation of HCI students involved in UCD projects and empirical data collected from industrial designers. Though simple, it models the designer's behavior and can be effectively used to (a) choose adequate tool support for a given phase of a project and (b) drive the development of new UCD tools.

There are many studies devoted to analyzing general software development practices and how to better support these. However, literature for qualitatively studying UI-related work practices in software development is relatively rare. We contribute to this body of knowledge by arguing the importance of supporting workstyle transitions in UI practices and presenting (i) a survey of 370 practitioners' answers to their current workstyles and tools' usage; and (ii) concrete examples in the form of design tools that try to support the most important workstyle transitions.

We also designed a framework aimed at studying the designer's behavior in a way that combines qualitative and quantitative data in a single model. The experiments we describe are not intrusive and allowed us to obtain statistically significant results.

The results suggest that workstyle transitions can influence the perceptions of use as well as intentions to use a given design tool. The implications of these findings for tool developers are also discussed.

Keywords:

Interaction Design, Usage-Centered Design, Software Development, UML, Usability, Software Engineering.

Resumo

Os modelos e métodos de Engenharia de Software (ES), assim como as suas ferramentas de suporte, têm ignorado os aspectos de design relacionados com a interacção com o utilizador (IU). E esta falta de usabilidade presente em muitas das ferramentas actualmente utilizadas conduziu a um baixo nível de satisfação com as ferramentas, tanto da parte de peritos de desenho de interacção como da parte dos engenheiros de software.

Durante esta investigação, observou-se que os aspectos de interacção com o utilizador, presentes no actual desenvolvimento de software poderiam ser melhorados se designers e engenheiros utilizassem ferramentas capazes de se adaptarem de forma transparente a qualquer estilo de trabalho.

Um novo modelo de estilos de trabalho para Desenho Centrado no Utilizador (DCU) é descrito. A proposta é nova por ser o primeiro modelo de estilos de trabalho especificamente desenhado para DCU. O modelo foi construído com base na identificação dos principais obstáculos entre DCU e SE, na literatura actual, na observação extensa de estudantes de IU envolvidos em projectos DCU e em dados empíricos recolhidos a partir de designers industriais. Apesar da simplicidade, o modelo captura o comportamento dos designers e pode ser utilizado para (a) escolher ferramentas de suporte adequadas a uma determinada fase de um projecto e (b) inspirar e estruturar o desenvolvimento de novas ferramentas DCU.

Existem muitos estudos dedicados a analisar as práticas gerais de desenvolvimento de software e como as suportar melhor. Contudo, a literatura sobre o estudo qualitativo das práticas relacionadas com IU é relativamente escassa. Contribuímos para esta base de conhecimento argumentando a importância do suporte às transições entre estilos de trabalho em IU e apresentamos (i) um estudo efectuado a 370 profissionais da área acerca dos seus estilos de trabalho e ferramentas utilizadas; e (ii) exemplos concretos sob a forma de ferramentas de design que suportam as mais importantes transições.

Também foi desenhado um modelo conceptual destinado ao estudo do comportamento do designer numa forma que combina dados qualitativos e quantitativos num único modelo. As experiências descritas não são intrusivas e permitiram a obtenção de resultados estatísticos significativos.

Os resultados sugerem que as transições entre estilos de trabalho podem influenciar as percepções de usabilidade de uma determinada ferramenta, assim como as intenções de utilização dessa ferramenta. As implicações destes resultados para os designers de ferramentas são apresentadas e discutidas.

Palavras-Chave:

Design da Interacção, Design Centrado na Utilização, Desenvolvimento de Software, UML, Usabilidade, Engenharia de Software.

Contents

List of Figures

List of Tables

Acknowledgements

Writing a dissertation about workstyles has been a rewarding opportunity for also reflecting upon my own workstyles. From that reflection, I found out that my scientific activity was often performed alone and at late hours or unconventional places. However, I was also frequently engaged into fruitful collaborative workstyles. Therefore, this dissertation was built out of multiple, valuable inspiration, helpful comments and many remarkable moments and talks, for which I feel deeply grateful.

First of all, I thank my advisor, Professor Nuno Jardim Nunes, who provided me with the guidance and challenges that I needed to conduct my research.

When I started this research, the first book I read was Larry Constantine and Lucy Lockwood's *Software for Use*. The influence of their work carried on constantly, and I was incredibly fortunate to have benefited from Larry's caring feedback, advice, and vision.

A warm thanks goes to all my DME colleagues, who affectionately welcomed me and have accompanied me ever since: in particular to Prof. Carmo, Eduardo Fermé, Sandra Mendonça, Elsa Carvalho and Leonel Nóbrega, who accompanied me more closely and always cared about my (sometimes unpredictable) workstyles.

I thank Prof. Falcão e Cunha and Lia Patrício for their care and support during the time I spent at Porto, for the interesting discussions and for the ideas they gave me.

This work was brought to light from many fruitful discussions, precious coffee and lively dinners at international conferences. I am particularly grateful to Annelise Mark-Pejtersen, Torkil Clemmensen, Rikke Orngreen and William Wong for trusting me the organization of a serious event. To Prof. Jean Vanderdonckt, for feedback on the workstyle model and for supporting and believing in this kind of work. To Prof. Oscar Pastor and Silvia Abrahão, for helping me tighten the concepts developed, and to Hallvard Traetteberg for sharing his ideas on modeling tools.

The students who participated in the tools' evaluation sessions, as well as all the anonymous users of the tools, significantly helped to improve my work. I am grateful for their clear insights, sharp comments and creative suggestions.

I also wish to thank to all the professionals that answered the survey, as well as to the IxDA and CHI-Web mailing list moderators for facilitating its dissemination. A special thanks goes to Grady Booch who kindly accepted to blog about this research and also disseminate the study.

I could have never made it here without the support from my family. I thank my sister Ana and my brother Miguel for being my best friends. I dedicate this thesis to my parents, Graça and Fernando, knowing that no action will ever be enough to thank them for the books, the journeys, the support and the advice, and for simply being the *best*.

Finally, I thank my fiancée Dalila, for her unconditional love and support, and for making my life look like a dream.

To my parents,

who have always encouraged me

1 Introduction

"Men have become the tools of their tools."

Henry David Thoreau, US Transcendentalist author (1817 - 1862)

The relationship between software engineering practitioners and their tools has always been a love-hate one. Because all software tools should ultimately serve human needs, the design of the interface between software and its users is a necessary and integral part of software engineering. Nevertheless, software engineering models and methods, as well as the tools supporting them, have largely ignored User Interface (UI) design issues. And the lack of usability present in many of the currently used design tools has lead both interaction designers and software engineers to a low level of satisfaction with their current tools of the trade. How can we expect that practitioners design and deliver usable systems if the design tools that they use themselves are not usable nor useful?

This question is apparently very timely: ACM Queue's "What's on your hard drive?" column[1], for instance, is testimony to the fact that tools influence the work of practitioners to the point where one really hates or loves a tool. Some practitioners even love *and* hate the same tool. IEEE Software's "Tools of the Trade" column also recently reported important reflections about software engineering tools and their usage (Spinellis, 2005).

In our research, we observed that the UI aspects of current software development could be relieved if both designers and developers used tools that could transparently adapt to any given style of work (what we call *workstyle*). We have found – through empirical observation in small software development settings, informal and formal usability studies and data obtained from surveys and questionnaires, as well as from current research literature –

[1] What's on your hard drive? – Tools that you both love and hate. *ACM Queue* 3 (5):10.

that both designers and software engineers often engage into different workstyles and move between workstyles very frequently (what we call a workstyle *transition*).

Despite there are many studies devoted to analyzing general software development practices, the literature for qualitatively studying UI-related work practices in software development is relatively rare. We try to increase this small but growing body of knowledge by arguing the importance of supporting workstyle transitions in UI practices and by presenting evidence of the importance of those transitions, the current tools employed by designers, and how they influence the tools' perceptions of usability.

The approach described in this thesis has been based on design research (March and Smith, 1995), which has an instantiation as final output. This instantiation should "operationalize constructs, models and methods". Therefore, we have (a) iteratively studied, captured and described UI-related behavior in a model and framework, (b) designed tools using that model for guiding and informing design decisions, and (c) used the revised model for evaluating the designed tools and improving on them.

1.1 Motivation

Over the past 20 years, the processing power of personal computers has increased by at least three orders of magnitude. Since the 80's, personal computers are 10,000 times faster with 10,000 times more memory and disk space. However, user interfaces remain essentially the same (Beaudouin-Lafon, 2004). And since user interfaces are built using tools which have not experienced any significant evolution over the years, the cycle continues.

Consider the kind of tool that is most specific, direct and widely spread for interface design: visual interface builders. For example, consider the tool employed by the author to design and develop the tools presented in this dissertation, Apple's Interface Builder (Apple, 2006), which is illustrated in Figure 1.1. As Apple's official description states, "it provides palettes, or collections, of user interface objects to an Objective-C [the native Mac OS X programming language] developer. These user interface objects contain items like text fields, data tables, sliders, and pop-up menus. Interface Builder's palettes are completely extensible, meaning any developer can develop new objects and add palettes to Interface Builder."

To build an interface, a developer simply drags interface objects from the palette onto a window or menu. Actions (messages) which the objects can emit are connected to targets in the application's code and outlets (pointers) declared in the application's code are connected to specific objects. In this way, all initialization is done before runtime, which improves performance and streamlines the development process.

What's interesting to note is that today's "modern" Interface Builder, as marketed and announced by Apple, remains essentially the same tool as the NEXT Step's 1988 Interface Builder. In fact, the original advertising campaign, from 1988, stated the following [taken from NeXT's brochure advertising the NeXT workstation (NeXT, 2006)]:

> *"Choose from a palette of interface objects in the Application Kit, like buttons, sliders and menus. Now you can resize and reshape objects and link those that relate to each other. (...) Arrange the interface so it looks exactly as you want the finished application to look. Even if you're a non-programmer, at this point you've successfully "prototyped" an application you can give to a professional".*

As Figure 1.1 demonstrates, apart from the modern, vivid and well-designed color schemes, the principles and concepts remained untouched and resisted the test of time which seems to run so fast in the Software Engineering industry.

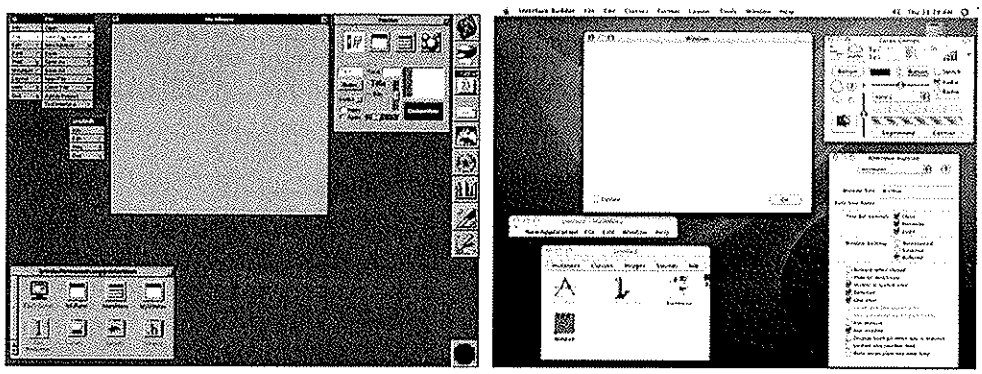


Figure 1.1 Left: NEXT Step's 0.8 Interface Builder (1988). Right: Mac OS X's Interface Builder latest release (2006). Apart from the appealing, highly aesthetic graphics, UI mainstream tools continue to offer basically the same features they offered 18 years ago.

However, modern design practices became more demanding, and because of the lack of evolution observed in visual interface builders, designers were forced to use other tools to perform their daily tasks. If we observe the current prototypical tools used by interaction designers, this is what we would find: Whiteboards, Paper and Pencil, Collaborative tools

such as Messenger and E-mail clients, and Analysis tools like MS Visio (see CHAPTER 4 for a survey on the practitioners' currently used UI tools).

Does this mean that there is a lack of research towards building more adequate and useful UI tools? Not at all, since a quick look at the current available literature reveals hundreds of different UI tools (Olsen and Klemmer, 2005). It's just that they simply never got adopted by interaction designers and software engineers.

This thesis provides new insight regarding the nature of the UI analysis and design practices. It is not merely a question of having more usable tools, but a matter of experience. It is very difficult to build a tool that efficiently covers all the aspects involved in UI analysis, design and implementation. Instead, what is needed is a set of tools that the designer can pick according to his or her current activity. The best designs support the task in a way that is simple and natural for the intended user. The interface itself becomes transparent; it allows the user to focus exclusively on the data being manipulated and on the progress towards the goal.

Consider the two different hammers shown in Figure 1.2. Each of them is a specialized version of the basic hammer for two different, but similar, tasks. When the tool-task relationship is mismatched, the user becomes "the tool of his tool", struggling to get the tool to achieve his goal. It's harder to pound nails using the claw hammer shown on the left side of Figure 1.2. For that task, the rubber mallet shown on the right side is more adequate. Therefore, having the right tool at the right time is the difference between satisfaction and accomplishment versus annoyance and frustration.

Figure 1.2 Specialized tools for different – but similar – tasks. On the left side, a claw hammer for assembling furniture and on the right, a soft rubber mallet for pounding nails.

However, switching tools is sometimes distracting and even difficult. It would be much more pleasant to use a tool that could be flexible and adjustable to more than one task or activity. In the hammer example, if the tool was smart enough to know whether the user

was pounding nails or assembling furniture, and if it adapted automatically to the "claw" or "rubber mallet" format, then the usage experience provided by the hammer would be reasonably improved.

The essence of this thesis lies in grounding foundations for supporting transparent, smooth adjustments to the activities and tasks being performed by interaction designers. It's important to have a wide range of tools very adequate to all the tasks encountered, but it would be even better if the tool itself could adjust to the way the user is working (his *workstyle*). It is not possible to build flexible, supportive UI tools without having a considerable understanding of the styles of work and activities involved in UI-related practices. Therefore, an essential part of this thesis is devoted to providing more insight into how tasks are actually performed, and which tools are being currently employed.

1.2 Problem Statement

The hypothesis underlying this research is the following: modeling the interaction designers' workstyles and using that information in the design of tools that can efficiently support *transitions* in those workstyles, will lead to (i) more useful and usable tools and (ii) to a better understanding of the design activities present in the life of interaction designers and software engineers engaged in interaction design activities.

We sustain our argument in the essence of Usage-Centered Design (UsageCD) practices: the lack of usability of interaction design tools is a problem that should be tackled using UsageCD techniques. Our approach is specifically targeted at software engineers engaged in interaction design activities.

1.3 Research Method

The research approach followed has been based on Design Research. In a thoroughly cited paper, March and Smith (1995) contrast the approach of design research with natural science research, and propose four general outputs for design research: constructs, models, methods, and instantiations. Constructs are the conceptual vocabulary of a problem / solution domain. Constructs arise during the conceptualization of the problem and are refined throughout the design cycle (see Figure 1.3). Since a working design (artifact) consists of a

large number of entities and their relationships, the construct set for a design research experiment may be larger than the equivalent set for a descriptive (empirical) experiment.

A method is a set of steps (an algorithm or guideline) used to perform a task. "Methods are goal directed plans for manipulating constructs so that the solution statement model is realized" (March and Smith, 1995). Implicit in a design research method is the problem and solution statement expressed in the construct vocabulary. In contrast to natural science research, a method may well be the object of the research program in design research. Since the axiology of design research stresses problem solving, a more effective way of accomplishing an end result – even or sometimes especially a familiar or previously achieved end result – is valued.

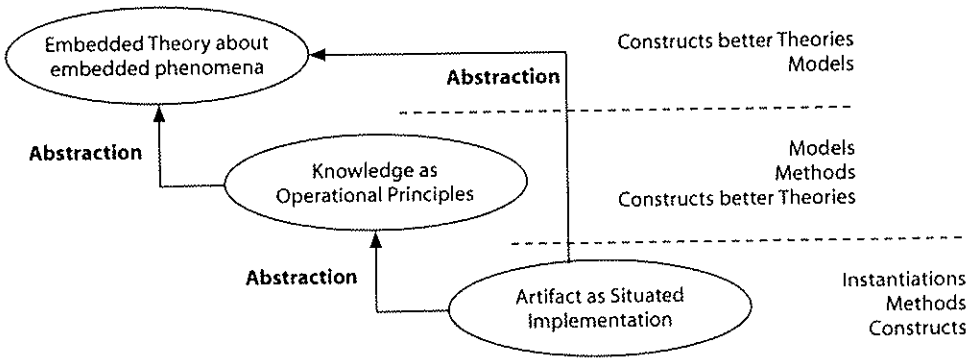


Figure 1.3 The outputs of Design Research [Source: isworld.org].

A model is "a set of propositions or statements expressing relationships among constructs." March and Smith (1995) identify models with problem and solution statements. They are proposals for how things are. Models differ from natural science theories primarily in intent: natural science has a traditional focus on truth whereas design research focuses more on (situated) utility. Thus a model is presented in terms of what it does and a theory described in terms of construct relationships. However, a theory can always be extrapolated to what can be done with the implicit knowledge and a set of entities and proposed relationships can always be expressed as a theoretical statement of how or why the output occurs.

The final output from a design research effort in March and Smith's explication is an instantiation which "operationalizes constructs, models and methods." It is the realization of the artifact in an environment. Emphasizing the proactive nature of design research, they point out that an instantiation sometimes precedes a complete articulation of the conceptual vocabulary and the models (or theories) that it embodies.

In the context of this engineering-based thesis, the design research we performed was based on studying and modeling UI-related behavior, designing tools using that model and using the revised model for evaluating the tools and improving on them. We began by creating an instantiation that preceded the establishment of a common vocabulary: the initial version of our CanonSketch tool (see § 5.2.1). Following the design research approach, we then studied and developed models and methods for abstracting, capturing and describing the interaction designer's workstyles (see CHAPTER 4). The CanonSketch tool was then refined, and the TaskSketch tool, as well as the final version of CanonSketch were designed in order to support the models presented in CHAPTER 4. As a final abstraction step, our research ended with the creation of an evaluation model for the tools developed (see CHAPTER 6).

1.4 Thesis Contributions

This thesis offers contributions in two inter-related areas:

A. New tools designed in order to support a set of workstyle transitions.

The tools developed and described in this thesis are specifically designed for smoothing transitions between workstyles. They have proven useful not only for teaching UCD concepts at University level, but also for industrial settings where they have been employed. These proof-of-concept applications aim at providing software engineers with tools that can help them build better UIs, and therefore this dissertation is a step towards bridging software engineering and human-computer interaction.

B. A new method for studying and evaluating UI-related tools.

Our framework can be used not only to evaluate tools but also to inform and guide the design of a new generation of tools. It is the first time that a model based on work-

styles and on workstyle transitions is defined and used in the UI field. Examples of usage for this new proposed model are given, and several tools are classified using the model. Concrete examples of how the model inspired the design of certain tool features are also provided.

The set of data collected during this research contributes to a small but growing body of knowledge about the software and interaction designers' work practices. More insight is provided regarding which tools are currently employed by professionals engaged in UI-related activities (through a survey made to 370 practitioners). And evidence that workstyle transitions *can* influence a tools' perceptions of use is also given. Moreover, and for the first time, techniques are provided for effectively combining quantitative workstyle-related data (such as switching frequency between different-detail views, or artifact's rate of change) with qualitative data (such as behavioral intentions to use a given tool). This combination of methods allows us to see a bigger picture of the tool's overall usage problems.

Other contributions of this dissertation are the following:

- It describes principles, methods and models for designing and evaluating interaction design and software development tools in terms that are adequate and relevant to the modern, fast-paced interactive systems developers and practitioners;
- It describes and reviews state of the art tools for supporting UI-related activities present in current, modern software engineering;
- It presents in detail the experiments, surveys, questionnaires and procedures that were undertaken during the evaluation of the tools built during this thesis, which can be reproduced and applied to similar contexts;
- It provides examples of UI patterns which are described, for the first time, at different but synchronized levels of detail, perspective, formality and functionality (using the developed tools). This multiple-level information can be used to promote reuse of UI recurrent solutions in a broad range of platforms, domains and development teams.

- It sets out some practical principles that arise from the design and evaluation process of the tools developed, exposing the benefits of designing to support workstyles and workstyle transitions in UI analysis and design.

1.5 Dissertation Roadmap

CHAPTER 2 describes the background work in which this thesis was built. § 2.1 describes and reflects upon the Usability process, under a software engineering perspective. § 2.2 surveys some of the models for capturing, describing and exploiting human work, a central piece of this dissertation. This includes cognitive work analysis, activity theory and a new research area, human work interaction design, which we also helped to initiate (Clemmensen et al., 2006). It is also important to study the approaches for Evaluation of tools and § 2.3 presents some methods, models and frameworks for evaluating current tools. Some of the methods, such as the Technology Acceptance Model (Davis, 1989) or Traeetteberg's cube (Traeetteberg, 2003) were deeply influential towards this dissertation. § 2.4 closes the chapter with some brief conclusions.

CHAPTER 3 overviews the state of the art in terms of interaction design tools. It focuses specifically on reviewing interaction design tools that are aimed at software engineers. § 3.1 introduces the theme; § 3.2 overviews Analysis, Design and Modeling tools; § 3.3 describes and discusses issues about Model-based UI Design tools; § 3.4 presents the latest tools focused on informal input techniques, such as gesturing and sketching; § 3.5 describes Collaborative design tools; § 3.6 reviews XML-based languages for UI modeling, as well as the tools supporting them; § 3.8 discusses issues about the future of UI tools and finally, § 3.8 draws some conclusions and insights about requirements for the next generation of interaction design tools.

CHAPTER 4, "Styles for Workstyles", presents our framework's foundation, including the rationale used to specify and design a new model for capturing UCD development practices as well as the designer's behavior. § 4.1 introduces the framework and models; § 4.2 describes and argues the importance and cost of workstyle transitions in interaction design-related activities; § 4.3 presents the workstyle model for UCD; the model is used to evaluate current tools and inspire design features in § 4.5; and it is also demonstrated how

it can be used to assess a UCD project's stage and effort in § 4.4. Since we were interested in assessing the real significance of workstyle transitions in a professional, real world context, § 4.6 describes a survey conducted among professional practitioners engaged in UI-related activities; § 4.7 outlines some possibilities about how the workstyle modeling technique can be applied to other problem domains; and § 4.8 concludes with the most important observations that were made during the elaboration of the workstyle model.

CHAPTER 5, "The Design of Design Tools", describes the design features of the two tools built during this research: CanonSketch (§ 5.2) and TaskSketch (§ 5.3), and how they were invented in order to support workstyle transitions. The discussion is illustrated with relevant examples and concrete case studies which prove the usefulness of both tools in analyzing, modeling and designing interactive systems. § 5.4 concludes the chapter by presenting some of the most relevant insights which were drawn from the tools' design experience.

CHAPTER 6 presents in detail the evaluation of the two tools. § 6.1 describes the evaluation study conducted to assess workstyle support in the CanonSketch, as well as its overall usability satisfaction rate. § 6.2 presents both the results for the TaskSketch tool as well as a more general framework for studying the designer's behavior and evaluating a given tool's level of workstyle support (and the influence of workstyle transitions in the user's perception of use when working with a given tool). § 6.3 is dedicated to reflecting and establishing some principles that came up during the evaluation sessions of both tools, and it clearly states some of the observations about which features worked and which didn't work. In § 6.4, the limitations of the study as well as its implications for tool developers are presented and discussed.

Finally, CHAPTER 7 draws some conclusions, limitations of the approaches presented and it also outlines future developments in the never-ending quest for more usable and useful UI tools.

2 Background

Models, Methods and Tools for Designing and Evaluating Design Tools

"Get the habit of analysis – analysis will in time enable synthesis to become your habit of mind."

Frank Lloyd Wright (1867 - 1959)

This chapter describes how existing models and methods for designing and evaluating interaction design tools have inspired and serve as foundation for the present work. The main issues described in this chapter are drawn from Human-Computer Interaction and Software Engineering literature.

The chapter begins by presenting the definition and main concepts of Usability from a software engineering perspective. SECTION 2.1 overviews the Usability Process and the importance of following it in order to pursue more usable products. The description of the usability process is divided among the analysis and design phases. Regarding the analysis phase, we briefly describe techniques for user and task analysis. The design phase description focuses on UI design principles, prototyping techniques, such as Wire-frames, Paper Mock-Ups, Storyboards and Scenarios, and Canonical Abstract Prototypes; and finally UI design evaluation methods, like Usability walkthroughs, Heuristic Evaluation, Thinking-aloud Protocols and collaborative Usability Inspections. As state of the art examples of Usability Processes, we briefly describe the Usage-Centered Design method (Constantine and Lockwood, 1999) and the Wisdom method (Nunes and Cunha, 2000). The section ends with a brief description and illustration of the value of using patterns for UI design, presenting patterns as building blocks for encapsulating UI design knowledge.

SECTION 2.2 is devoted to surveying models for capturing and describing human work, a central theme of this thesis. This theme was based on IFIP's new working group 13.6,

which we helped create. This new working group is dedicated to studying Human Work Interaction Design, and its goals, intentions, scope and plans are described in this section. As a sequence to the Human Work Interaction Design description, we then describe workstyle models for general software development and a representation framework for user-centered design. Cognitive Work Analysis, a work-centered conceptual framework is also surveyed in this section, followed by Activity Theory, a field which has dominated much research approaches in HCI.

SECTION 2.3 is dedicated to covering the final aspect of this thesis, which consists not only on designing new tools but also evaluating them. Therefore, in SECTION 2.3 we describe models and methods for evaluating tools. These include the Technology Acceptance Model and related models, empirical studies on programming, software design and development tasks, and the Cognitive Dimensions of Notations, a technique for evaluating any kind of notational system, such as a visual programming language, or any kind of digital tool.

2.1 On Usability for Software Development

Usability engineering provides important benefits in terms of cost, product quality, and customer satisfaction. It can improve development productivity through more efficient design and fewer code revisions. It can help to eliminate over-design by emphasizing the functionality required to meet the needs of real users. Design problems can be detected earlier in the development process, saving both time and money. It can provide further cost savings through reduced support costs, reduced training requirements, and greater user productivity. A usable product means more satisfied customers and a better reputation for the product and for the organization that developed it.

Today, many leading corporations such as Eastman Kodak Company, American Airlines, Apple Inc., Lotus Development Corporation, and Microsoft Corporation are incorporating usability engineering into their product development cycles. For them, usability is becoming a competitive advantage.

2.1.1 Usability Concepts

The Usability of a product is defined in the ISO 9241, Part 11, as "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a given context of use" (ISO, 1998).

The term Usability Engineering is used to ensure that the software products developed actually achieve that level. The term was coined to reflect the traditional engineering approach some usability specialists adopt (Good et al., 1986). It is "a process through which usability characteristics are specified, quantitatively and early in the development process, and measured throughout the process" (Hix and Hartson, 1993). It is a multi-disciplinary issue that can be approached through many different perspectives, such as sociology, psychology, visual arts and of course computer science.

Usability is typically measured by testing a number of representative users performing a predetermined set of tasks. To determine the system's 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).

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; this is usually determined by measuring the time a user spends working with the system before she can complete certain tasks in the time it would take an expert to complete those same 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; As Ferré et al. (2001) describe, "the higher the system usability, the faster the user can perform the task(s) and complete the job";
- User Retention over time – the system should be easy to remember, allowing casual users to reuse the system without having to learn the system again; this attribute reflects how well the user remembers how the system works after a period of non-usage;
- 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; this attribute does not include system errors, it addresses only the errors made by the user; a usable system is the opposite of an error-prone system;
- Satisfaction – The system should be pleasant to use, fostering subjective satisfaction in use.

As Ferré et al. (2001) point out, one problem regarding usability is that "these attributes sometimes conflict. For example, learnability and efficiency usually influence each other negatively. A system must be cautiously designed if it requires both high learnability and high efficiency." As an example, they refer that using accelerators (a combination of keys to perform frequent task(s)) usually solves this clash.

2.1.2 *The Importance of Usability*

Nowadays, the need for usability is very real, and its importance is well-recognized. In general, both consumers and technology companies have accepted the notion that if a

product is easy to use, it will probably sell more and it will also require less maintenance. However, this wasn't the case in the early days of software-intensive systems. But today, usability specialists are paid to ensure that software is practical and usable.

Usability comes with a price, and usually it does require an investment. It costs money to provide staff, training, standard, tools and a user-centered design process. However, cutting training costs by 75 percent may be on the high end of the savings one typically gets from usability (Nielsen, 1993).

Jakob Nielsen, one of the leading experts on usability, suggests that 10% of a website development budget should be spent on usability and that this investment doubles the usability of the redesigned site. On average, Nielsen says that this usability improvement leads to the following improvement in site statistics:

- 100% improvement in sales or conversion rates
- 150% improvement in traffic and visitor counts
- 161% improvement in user performance and productivity
- 202% improvement in use of specific site features

The biggest gains often come from savings on subsequent maintenance costs, since most requests for enhancements are because of a poor match between the original system's design and the users' actual needs. But proper usability design commonly cuts training costs by 50 percent and increases productivity by 25 percent. And there's one additional benefit that usability engineering practices can present for projects: employees may actually start liking their company's internal software.

2.1.3 The Usability Process

Since all software serves human needs, the design of the interface between software and its users is an integral and important part of software engineering. Designing this user interface requires the gathering of information regarding (i) who are the system's users; (ii) what tasks do they need to accomplish and (iii) what does the system needs to provide them in order for them to complete their tasks efficiently (Constantine and Lockwood, 1999).

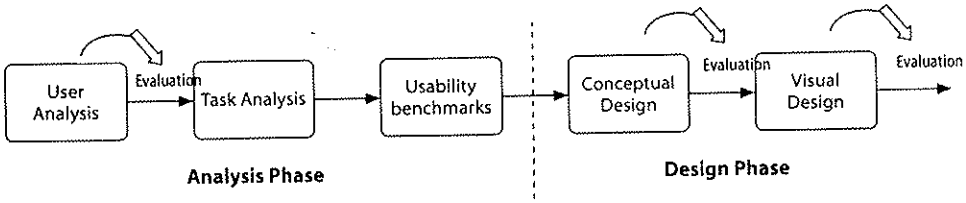


Figure 2.1 The Usability Process.

The usability process aims at helping interaction designers and software engineers answer these questions during the analysis phase and support the design during the following design phase. The process is illustrated in Figure 2.1 and we will describe the phases in a generic, method-independent way in order to focus on the issues that are common to all usability processes that exist. In subsections 2.1.8 and 2.1.9, respectively, we will then briefly describe two methods for integrating usability in the software engineering process, Usage-centered Design (Constantine and Lockwood, 1999) and the Wisdom method (Nunes and Cunha, 2000, 2001).

2.1.4 *Analysis*

In the analysis phase, usability experts work towards getting to know the users and their needs, expectations, interests, responsibilities and contexts of use. This phase can be subdivided into user analysis and task analysis, both of which we will briefly overview.

User Analysis. Typically, the user analysis' source of information in this phase is composed of field visits. Other methods commonly used are focus groups, surveys and derived data. Usage-centered Design (Constantine and Lockwood, 1999) proposes Structured Role Models, as a way to collect and organize information about users. A user *Role* is defined by the characteristic needs, interests, expectations, behaviors and responsibilities a user assumes, in a given instant, with the system. A user Role has a name, context (describing the overall responsibilities and the larger context), characteristics (interaction patterns, behaviors and attitudes) and a set of criteria (special objectives or special needs that have to be addressed during that role). Considering a designer-modeling role, for instance, the context would be "problem-solving oriented, under pressure, deals with multiple stakeholders". The characteristics could include "computer-literate, expert; fast or incremental

changes to models". And finally, criteria would focus on needs for multi-dimensional, traceable modeling capabilities.

In field visits, developers watch and observe target users performing their tasks in their working environment, using either the system to be replaced or performing them manually, if there is no existing tool. In addition, developers interview the users to capture and understand their motivation and the strategy and rationale behind their actions. In particular, the method of context inquiry (Beyer and Holtzblatt, 1998) has gained popularity among practitioners. It is a specific type of interview for gathering field data from users. It is usually done by one interviewer speaking to one interviewee (person being interviewed) at a time. The aim is to gather as much data as possible from the interviews for later analysis. It offers a structured way for gathering and organizing this information.

Focus groups are a powerful means to evaluate services or test new ideas. Basically, focus groups are interviews, but of 6-10 people at the same time in the same group. Powell et al. (1996) define a focus groups as "a group of individuals selected and assembled by researchers to discuss and comment on, from personal experience, the topic that is the subject of the research."

One can obtain a great deal of information during a focus group session. It is well adequate to acquiring several viewpoints and perspectives about the same topic as well as to gain insight into people's understanding of everyday system use.

In surveys, the quality of the information gathered depends very much on the quality of the questions asked. They are a one-way source of information, because it is often difficult or even impossible to verify and clarify the responses of the respondent. This is why this method is often used in conjunction with follow-up interviews (conducted with a significantly smaller sample of respondents. Dillman, one of the most well-known researchers in the field of survey-based research, describes the use of several implementation elements to achieve high response rates. In 1978, he developed a general method of implementation, known as the Total Design Method, which is known to achieve high response rates. Since then, Dillman expanded this design and re-named it *The Tailored Design Method* (Dillman, 1999). This method is very useful and gives explicit recommendations that we tried to follow in our research, such as designing a respondent-friendly questionnaire, shortening the

questionnaire (asking fewer questions to reduce respondent burden) and creating questions that the respondent will find interesting to answer.

Task Analysis. Task analysis is intended to describe a set of techniques people use to get things done. The importance of tasks during the whole product development cycle comes from the fact that focusing on a small set of tasks helps prioritize, negotiate and rationalize the implementation and development efforts. Constantine and Lockwood (1999) propose to cluster and rank the set of tasks by importance and frequency, in order to obtain a small set of tasks. They argue that this task-based approach guarantees that the implemented system will provide support for the most important functionalities, and prevent "creeping featuritis", i.e. situations in which features are added that do not quite add up to a complete and useful set of capabilities, leading to complex systems in which many of the features go unused by users.

After decomposing tasks into subtasks and subtasks into particular actions that the user might execute, it is possible to instantiate them to real world examples. This instantiation will allow the usability engineer to present and test with users in usability test sessions.

In the software development context, and in particular in the context of UI development, merely asking users what tasks they perform and how they perform them is not enough to allow us to design good programs. Users can answers us, with all their honesty, what they believe it's what they do in their quotidian working endeavors, but if we watch them at work, we will be lead to conclude that they perform different things than the ones they reported us.

Learning to design good programs requires a team effort between analysts and users, so that parties evolve towards a common, concise understanding of the work that should be supported by the product to be designed. That's the goal of task models which represent the user needs' structure: the system's architecture of use (Constantine, 2003).

There are many ways of modeling the use of a system. From flow diagrams (which describe tasks in terms of sequential events) to scenarios (which represent narratives of one or more activities).

In structured use cases, the system's usage scenario narrative is divided into two sections: the user's actions model, which describes what the user does, and the system's answer model, which describes how the system reacts to the user's behavior.

The following table shows a classical example of a structured use case for the task of obtaining money from an ATM machine:

Getting Cash User Intentions	System Responsibilities
Insert Card	
	Read card Request PIN
Enter PIN	
	Verify PIN Display transaction option menu
Press key	
	Display account menu
Press key	
	Prompt for amount
Enter amount	
	Display amount
Press key	
	Return card
Take card	
	Dispense cash
Take cash	

Structured use cases typically contain many pre-definitions (either implicitly or explicitly) about the shape of the UI which is still to be designed. Therefore, they tend to be much too close to the system's implementation detail, and if one bases on a too concrete use case that contains implicit design decisions, one runs the risk of losing many chances of obtaining a good design, a design focused on the users and not on the computer, platform, programming language, etc.

This lead to the invention of essential use cases, which are defined the following way:

An essential use case is "a single, discrete, complete, meaningful, and well-defined task of interest to an external user in some specific role or roles in relationship to a system, comprising the user intentions and system responsibilities in the course of accomplishing that task, described in abstract, technology-free, implementation-independent terms using the language of the application domain and of external users in role" (Constantine and Lockwood, 1999).

An essential use case is structured into three parts: a sentence describing the user's express intention, and a narrative divided into two parts: the user intentions model and the system's responsibilities model.

The difference between an essential use case and a structured one can seem subtle but its consequences are not. Constantine describes the following example, which consists in re-writing the previous use case in the essential way, by focusing on the user's intentions and not in the actions, and also simplifying and generalizing from the concrete case.

We start out by interrogating ourselves about the reason why a user should insert such a valuable card inside an ATM machine. It could even be regarded as a ridiculous action, especially if one takes into account that a machine can fail and keep the card inside, never returning it back. However, millions of users perform this task every day. So the question posed is: "What for?" And the answer is: so that he (the user) identifies himself to the ATM machine. Users do this because they don't want anybody else removing cash from their accounts.

The original case could be rewritten in the following essential version:

Getting Cash User Intentions	System Responsibilities
Identify self	Check identity Offer choices
Choose	Dispense cash
Take cash	

This essential use case is dramatically shorter and simpler than the concrete one (for the same interaction). This is because it includes only the steps that are essential and of genuine interest to the user.

Since it is problem-oriented, and not solution-oriented, it deliberately leaves many possibilities open in relation to the system's design and implementation. For instance, besides magnetic cards, there are many other ways to identify a user in front of a system. The ATM machine could examine the iris or the fingerprint to find out who is using the system. Another possibility left open by the essential use case is to offer choices through voice or confirm them through speech recognition and processing.

In conclusion, essential use cases are brief and avoid unnecessary debate about implementation details. There are also deeper advantages that relate to many aspects of system development. Most importantly, the abstraction of essential use cases comes from identifying user intentions and system responsibilities, and these act as heuristics that benefit software development.

2.1.5 Design

During the *conceptual design* phase, the basic user-system interaction and the objects in the UI are defined. This also includes the context in which the interaction takes place.

Some reference books that include general design principles that are adequate and interesting for any interaction designer or software engineer are *The Design of Everyday Things* (Norman, 1990), *The Inmates Are Running the Asylum* (Cooper, 1999) and *Designing Web Usability: The Practice of Simplicity* (Nielsen, 1999). For a different (and fun) perspective, see *GUI Bloopers* (Johnson, 2000).

Although there is a vast body of knowledge regarding design and design principles, it is essentially a very creative process, and there is no "cookbook" for automating this activity. Also, and since conceptual design defines the groundwork for the entire system, it is the most crucial piece in the process.

Some of the main principles for good UI design include (Constantine and Lockwood, 1999):

- *Feedback*: the design should 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;
- *Reuse*: the design should 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;
- *Simplicity*: the design should 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;

- *Structure*: the design should organize the user interface purposefully, in meaningful and useful ways based on clear, consistent models that are apparent and recognizable to users, putting related things together and separating unrelated things, differentiating dissimilar things and making similar things resemble one another. The structure principle is concerned with the overall user interface architecture;
- *Tolerance*: the design should 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 reasonable;
- *Visibility*: the design should keep all needed options and materials for a given task visible without distracting the user with extraneous or redundant information. Good designs don't overwhelm users with too many alternatives or confuse them with unneeded information.

After the conceptual design (and in some processes in parallel), comes the *visual design* phase (sometimes also called presentation or graphic design). This covers the handling of everything related to the UI's appearance.

Reference readings for visual design include *About Face* (Cooper and Reimann, 2003), and *Designing Visual Interfaces* (Mullet and Sano, 1995).

In *Designing Visual Interfaces*, the authors describe the underlying principles of visual design in the context of UIs, focusing on screen and graphics from a communication-oriented perspective, based on graphic design, industrial design and architecture. The practical techniques, common errors and thoroughly illustrated design examples are divided and approached through six major areas: elegance and simplicity, scale, contrast and proportion, organization and visual structure, module and program, image and representation and finally style (Mullet and Sano, 1995). To Mullet and Sano, "communication-oriented visual design views these forces not as irreconcilable opponents, but as symbiotic components of every high-quality solution." As they mention, "good graphic design can significantly improve the communicative value of the interface, leading to increased usability."

2.1.6 Prototyping

A prototype is a working model built to develop and test design ideas. In web and software interface design, prototypes can be used to examine content, aesthetics, and interaction techniques from the perspectives of designers, clients, and users.

The entire idea behind prototyping is to cut down on the complexity of implementation by eliminating parts of the full system. Horizontal prototypes reduce the level of functionality and result in a user interface surface layer, while vertical prototypes reduce the number of features and implement the full functionality of those chosen (i.e. we get a part of the system to play with). Figure 2.2 illustrates this commonly used classification space for the different types of prototypes.

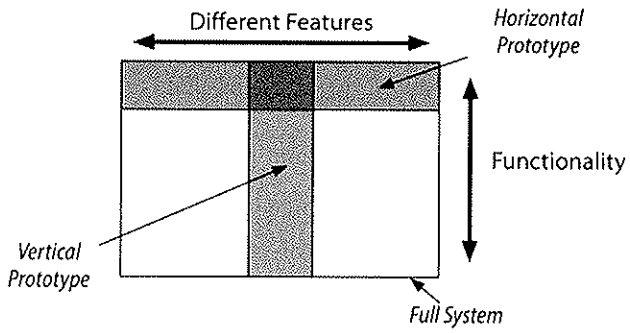


Figure 2.2 Classification Space of Prototypes.

Prototypes can also be classified according to their fidelity. Low-fidelity prototypes are quick, cheap, and designed to elicit user feedback as early as possible.

High-fidelity prototypes are more expensive and usually involve coding, but are better for evaluating graphics and getting 'buy-in' that usability problems found during testing are not due to the 'rough' quality of the prototype.

Walker et al. (2002) performed research aimed at indicating what level of fidelity and media could produce the best feedback from users. Their experiment compared user testing with low- and high-fidelity prototypes in both computer and paper media. Task-based user tests of sketched (low-fidelity) and HTML (high-fidelity) website prototypes were conducted in each medium, separating the testing medium from other factors of prototype fidelity. Walker et al. (2002) found that low- and high-fidelity prototypes were equally good at uncovering usability issues. Usability testing results were also found to be inde-

pendent of the medium (computer or paper). However, their users rated the prototypes as different in "professionalism," "finishedness" and "likeliness to change."

In fact, presenting a prototype representation that is too polished encourages clients to focus on irrelevant details such as fonts, colors, and images, when it is often desirable at this point to get feedback on the structure and organization of information (Wong, 1992). However, presenting too rough a representation can seem unprofessional and unimpressive. For design firms working with new clients, it is often important that they make a positive impression early in the design process to reinforce that the client made a good decision in hiring the firm. Early presentations must strike a delicate balance between keeping the focus on basic, structural issues and making a good impression.

We will overview the following prototyping techniques: paper mock-ups, scenarios, navigation maps, storyboards, schematics and canonical abstract prototypes.

Paper mock-ups. At the beginning of the design process, the designer usually sketches and creates paper prototypes (Wagner, 1990), which are usually composed of pencil drawings, printouts of screen designs, or a combination of the two. The designer himself acts as the computer system and shows the user the next element when a transition occurs (Preece et al., 1994).

Using common office supplies, (markers, index cards, transparency film, etc.) the development team can quickly construct a fully-functional prototype of the product interface. This technique has proven effective even for sophisticated, high-tech products.

Usability testing of a paper mock-up is quite straightforward. Using their finger as a "pointing device," users can select from menus, click on buttons, and otherwise interact with interface elements. One or two members of the development team simulate the behavior of the computer, taking the appropriate action in response to the users' requests. Paper mock-ups are easy to modify, so the team can even make changes in the middle of a usability test.

An iterative development process that employs low-fidelity prototyping is an excellent method for ending "opinion wars" and other project-killers. Instead of wasting time on subjective arguments, the development team can instead focus on objective usability goals (e.g., an untrained user can complete the installation within 5 minutes) and test different

approaches until the first satisfactory one is found. At that point, the team can turn their attention to the next issue on the priority list.

An electronic prototype (such as one developed in Visual Basic) seems more attractive at first glance because it has a more polished "look". However, when it comes to prototyping the "feel" of the product, i.e., its behavior, some form of programming is generally required. In a paper mock-up, the "feel" is simulated by the person playing the computer, so this programming time is eliminated. In our experience, most of the risky issues in product development pertain to the "feel" (Does the product have the right set of functions? Can users figure out how to do their work?) rather than the "look" (Are the icons clear? Does it conform to the style guide?).

Paper mock-ups are not a panacea. For example, it is not possible to assess response times with a paper mock-up, and it may be harder to get accurate feedback on the visuals of the product. Also, sometimes it is necessary to fake data that would be available in an actual running system.

Scenarios. Scenarios take prototyping to the extreme by reducing both the level of functionality and the number of features. By reducing the part of interface being considered to the minimum, a scenario can be very cheap to design and implement, but it is only able to simulate the user interface as long as a test user follows a previously planned path.

Carroll's (1995) definition of scenario states what a good scenario should cover:

"The defining property of a scenario is that it projects a concrete description of activities that the user engages in when performing a specific task, a description sufficiently detailed so that design implications can be inferred and reasoned about. Using scenarios in system development helps keep the future use of the envisioned system in view as the system is designed and implemented; it makes use concrete – which makes it easier to discuss use and to design use." (p. 4).

Since the scenario is small, we can afford to change it frequently, and if we use cheap, small thinking aloud studies, we can also afford to test each of the versions. Therefore scenarios are a way of getting quick and frequent feedback from users.

Scenarios can be implemented as paper mock-ups or in simple prototyping environments that may be easier to learn than more advanced programming environments (Niel-

sen, 1999). This is an additional savings compared to more complex prototypes requiring the use of advanced software tools.

Navigation Maps. Navigation maps are essentially a diagram specifying how the different interaction spaces (screens, dialog boxes, windows, etc.) are interconnected and how can the user flow through the UI during the course of his or her tasks. For instance, a navigation map could specify how parts of the user interface will be organized into tabbed notebooks accessed through a set of command buttons on a central dispatch dialog, with sections within notebook pages reached through shortcuts arrayed in a menu-like bar across the top of each notebook page.

Navigation maps often evolve throughout the entire life of the project, being updated constantly to reflect new understandings of the site structure. Early in the design process, navigation maps will reflect the application's structure broadly and, as time progresses, they will be revised to become increasingly detailed. They can then be used to support project management, content management, and the generation of specifications. Navigation maps are the primary artifact of information design, and in organizations that have information design specialists, the map will be generated and updated by that specialist. Navigation maps usually consist of labeled blocks and lines, with some additional features to indicate groupings.

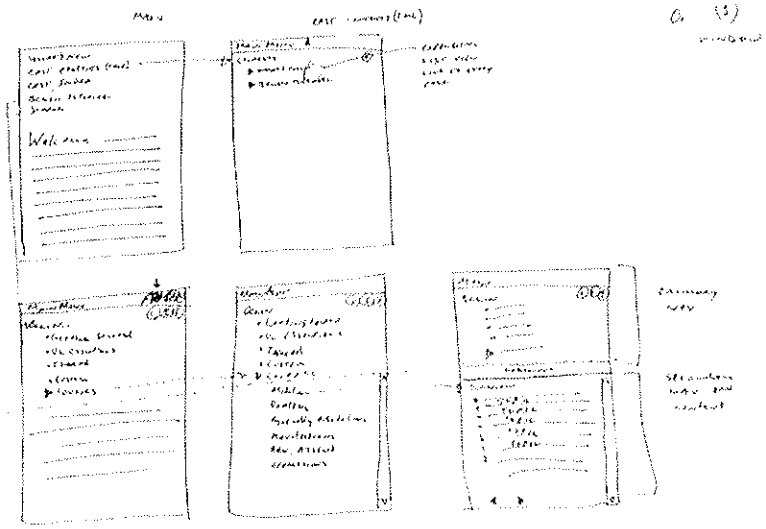


Figure 2.3 An example of a storyboard from a real world project [taken from (Newman and Landay, 2000)].

Storyboards. A storyboard is a representation of a particular interaction sequence. It is accompanied, either explicitly or implicitly by a narrative about the task the user would be trying to accomplish via the particular sequence depicted.

Storyboards reflect limited detail about the contents of each page in the sequence and only the navigation links required to accomplish the task are represented. For example, the storyboard shown in Figure 2.3 shows an interaction sequence that a user might execute in order to access information within a tutorial system. It shows what would happen if a user started at the main page, clicked "Begin Tutorial," then clicked "Courses," and then clicked "Modeling." Another possible sequence of user actions is shown: when the user clicks "Cast Contents" she will be presented with a table of contents. It is clear that there are links on several of the pages depicted that would lead to other pages, but those interactions are not shown (Newman and Landay, 2000).

Schematics. Schematics are representations of the content that should appear on a particular page. They are usually devoid of images, though they may indicate with a label where an image should be placed. While schematics are not meant to show how color, typography, and graphics will be used on the page, they may themselves use simple color (often they are monochrome or grayscale), typography, and graphics to indicate other things about the UI screen.

Wire-frame mockups are another kind of schematics, and an example is shown in Figure 2.4. The wire-frame represents the relative size and position of visual user interface elements. Color-coding of the areas may also be used to indicate the type of element represented or the relative importance or priority of the information or function. This latter variation has enjoyed some popularity among graphic designers for Web-based applications.

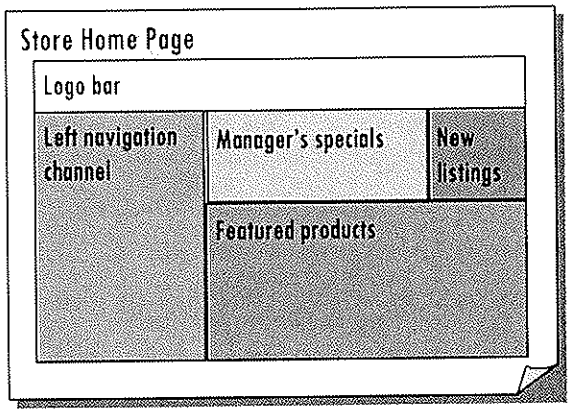


Figure 2.4 Wire-frame schematics.

Canonical Abstract Prototypes. Prototyping techniques still leave a considerable gap between the inception level models of user intentions (task cases, use cases, scenarios and other requirements level models) and the concrete user interface. The center ellipse in Figure 2.5 illustrates this gap. A growing awareness of this conceptual gap lead Constantine and colleagues to develop a new language for visual and interaction design, called Canonical Abstract Prototypes (Constantine, 2003).

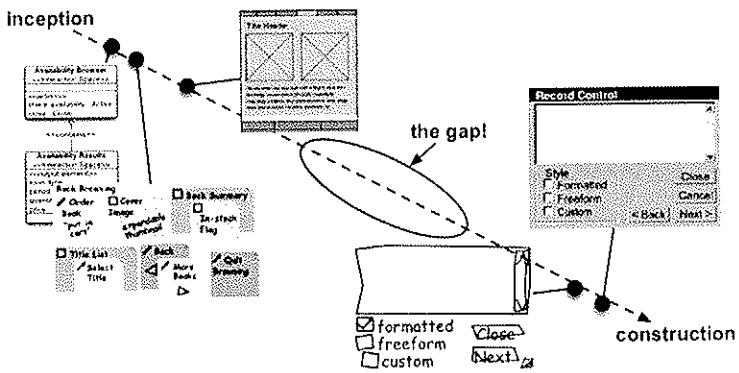


Figure 2.5 Prototyping techniques from inception to construction (adapted from (Constantine, 2003)).

This language fills the gap between existing inception level techniques, such as the illustrated UML-based screen stereotypes or visual content inventories, and construction level techniques such as concrete prototypes. At the heart of the approach is a standard, all-purpose set of abstract user interface components - canonical components -from which

the modeler constructs the abstract prototype. These canonical components are specific as to purpose - such as, contain, collect, select, create, move, and the like -without specifying the actual user interface widget for implementation. In all, just 15 abstract components are enough to model virtually any user interface in the abstract. The abstract materials in this "canonical set" include: container, collection, element, and notification; the abstract tools include: operation/action (generic abstract tool), start, quit, select, create, delete, modify, move, duplicate, accept, and link.

MATERIALS

SYMBOL	INTERACTIVE FUNCTION	EXAMPLES
	container*	Configuration holder, Employee history
	element	Customer ID, Product thumbnail image
	collection	Personal addresses, Electrical Components
	notification	Email delivery failure, Controller status

Figure 2.6 Symbols, function description and examples of Abstract Materials. [Taken from (Constantine, 2003)].

TOOLS

SYMBOL	INTERACTIVE FUNCTION	EXAMPLES
	action/operation*	Print symbol table, Color selected shape
	start/go/to	Begin consistency check, Confirm purchase
	stop/end/complete	Finish inspection session, Interrupt test
	select	Group member picker, Object selector
	create	New customer, Blank slide
	delete, erase	Break connection line, Clear form
	modify	Change shipping address, Edit client details
	move	Put into address list, Move up/down
	duplicate	Copy address, Duplicate slide
	perform (& return)	Object formatting, Set print layout
	toggle	Bold on/off, Encrypted mode
	view	Show file details, Switch to summary

Figure 2.7 Symbols, function description and examples of Abstract Tools [Taken from (Constantine, 2003)].

ACTIVE MATERIALS

SYMBOL	INTERACTIVE FUNCTION	EXAMPLES
	active material*	Expandable thumbnail, Resizable chart
	input/accepter	Accept search terms, User name entry
	editable element	Patient name, Next appointment date
	editable collection	Patient details, Text object properties
	selectable collection	Performance choices, Font selection
	selectable action set	Go to page, Zoom scale selection
	selectable view set	Choose patient document, Set display mode

Figure 2.8 Symbols, function description and examples of Abstract Active Materials (or Hybrids) [Taken from (Constantine, 2003)].

Figure 2.6, Figure 2.7, and Figure 2.8 detail the complete set of Canonical Abstract Components along with examples. As the dozen abstract tools in Figure 2.6 illustrate, interactive functions are distinguished from the perspective of users in interaction with a user interface. Using Canonical Abstract Prototypes, the transition from task model to design is reduced to two relatively straightforward translation processes, each of which addresses a limited set of specific decisions and issues. Although the design process is thus made substantially more orderly and manageable, the role of creative problem solving has not been eliminated – systematization is not equivalent to mechanization.

In contrast with schemes for automatically generate user interfaces from specifying models (Pawson and Mathews, 2002; Molina et al., 2002), canonical abstract prototyping "recognizes the pivotal role of human creativity and invention in designing good user interfaces. The use of canonical abstract prototypes merely serves to focus the attention and creative energies of the designer on matters of importance in the design task at hand" (Constantine, 2003).

An abstract prototype based on canonical components is just about half way between a task model based on essential use cases and a realistic paper prototype. For this reason, it smoothes and simplifies task-driven user interface design. Not only is it easier for the designer to pick appropriate abstract components based on task requirements, but each canonical component typically translates more or less straightforwardly into a small set of standard solutions. Less experienced designers are helped to quickly find good conventional solutions; sophisticated designers will find that common patterns and opportunities

for innovation are easier to recognize in models constructed from standard abstract components.

2.1.7 Evaluation

One of the most central and important activities in usability engineering is the evaluation phase. Finding usability defects and problems early, efficiently, and cost-effectively is an ongoing challenge for usability experts. In this section, we will review and present some of the most widely employed methods for conducting evaluation of an interactive system. We will review User Testing, Thinking-aloud protocols (Boren and Ramey, 2000), Cognitive Walkthroughs (Dix et al., 1993), Pluralistic Usability Walkthroughs (Bias, 1994), Heuristic evaluation (Nielsen, 1992) and Collaborative Usability Inspections (Constantine and Lockwood, 1999).

User Testing. The most effective usability testing, whether in a lab or in the field, answers specific questions through carefully devised test scenarios. Among the most effective uses for usability testing are:

- Comparing alternative solutions where the most conventional "best" approach is unclear;
- Evaluating a novel solution, or a solution that breaks conventional standards;
- Resolving design disputes over the effectiveness of one solution versus another;
- Evaluating the effectiveness of the UI's navigation: can users find what they are looking for or complete what they are trying to accomplish?

Figure 2.9 shows a typical laboratory of usability testing, with a view from the control room into the two test rooms. Test room ceilings carry dome cameras with remote pan-tilt-zoom-focus which can be positioned in any of 11 locations. Consultants can run sessions independently on each side. Alternatively, consultants can watch participants collaborate "remotely" across the two test rooms.

State of the art usability laboratories for conducting user testing feature a wide range of digital paraphernalia, such as dome cameras with inbuilt, remotely controlled, silent pan-tilt-zoom capabilities, and AV recording, analysis and editing equipment.

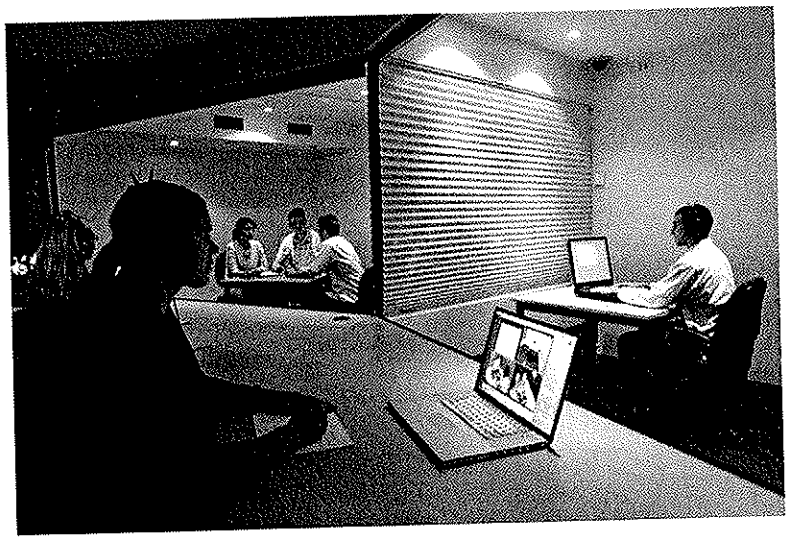


Figure 2.9 The usability laboratory at the University of Queensland, Australia.

Thinking aloud. The thinking-aloud protocol (Boren and Ramey, 2000) is another popular method for usability testing. In this protocol, a user is asked to perform a given task or set of tasks that were previously defined as being part of a test scenario (see § 2.1.6 for a definition of a scenario), and during the course of the user's test, the usability expert asks the user to vocalize his or her thoughts, opinions, feelings and impressions about the interaction and usage of the product being tested.

Thinking aloud allows us to understand how the user interacts with the system and what considerations the user mentally records when using the UI. If the user finds that the succession of steps required to complete the task's goal is different from what the user expected, perhaps the interface is complicated and difficult to use.

There are other benefits that the usability expert can achieve through the use of thinking aloud protocols, although their main advantage is a more concise understanding of the user's mental model and how the UI fits into the user's task accomplishment. Other benefits include, for example, the terminology that the user uses to express an idea or function: that terminology should be incorporated into the product design or at least its documentation.

Rubin (1994) provides practical, step-by-step guidelines for implementing thinking-aloud protocol tests, in a "liberally peppered book with real-life examples and case histories

taken from a wide range of industries". Nielsen (1992) describes a simplified version of thinking aloud protocols, arguing that traditional thinking aloud studies are conducted with psychologists or user interface experts as experimenters who videotape the subjects and perform detailed protocol analysis, which is costly and time-consuming. He shows that it is possible to run user tests without sophisticated labs, simply by "bringing in some real users, giving them some typical test tasks, and asking them to think out loud while they perform the tasks". Nielsen (1992) shows that computer scientists are indeed able to "apply the thinking aloud method effectively to evaluate user interfaces with a minimum of training", and that "even fairly methodologically primitive experiments will succeed in finding many usability problems", a philosophy that is part of his Discount Usability Engineering approach (Nielsen, 1993).

Besides reducing the number of subjects, another major difference between simplified and traditional thinking aloud is that data analysis can be done on the basis of the notes taken by the experimenter instead of using videotapes. Recording, watching, and analyzing the videotapes is expensive and takes a lot of time which is better spent on running more subjects and on testing more iterations of redesigned user interfaces. Video taping should only be done in those cases (such as research studies) where absolute certainty is needed.

Interestingly enough, the think-aloud method can also be used in the study of reading. Readers are asked to "think aloud" while reading in order to establish what inferences they are drawing from a text. In the development of reading tests, this method may be used to emphasize areas of questioning.

Cognitive Walkthrough. Cognitive walkthrough (Dix et al., 1993) involves one or a group of evaluators inspecting a user interface by going through a set of tasks and evaluating its understandability and ease of learning. The user interface is often presented in the form of a paper mock-up or a working prototype, but it can also be presented as a fully functional interface. The input to the walkthrough also includes the user profile, especially the users' knowledge of the task domain and of the interface, as well as the task cases. The evaluators may include human factors engineers, software developers, or people from marketing,

documentation, etc. This technique is best used in the design stage of development. But it can also be applied during the code, test, and deployment stages (Wharton et al., 1994).

The code walkthrough approach to evaluation, which is familiar in the software engineering field, is the root of the cognitive walkthrough approach. Walkthroughs require a detailed review of a sequence of actions. In the code walkthrough, the sequence represents a segment of the program code that is stepped through by the reviewers to check certain characteristics (e.g., that coding style is adhered to, conventions for spelling variables versus procedure calls, and to check that system wide invariants are not violated).

In the cognitive walkthrough, the sequence of actions refers to the steps that an interface will require a user to perform in order to accomplish some task. The evaluators then step through that action sequence to check it for potential usability problems. Usually, the main focus of the cognitive walkthrough is to establish how easy a system is to learn. More specifically, the focus is on learning through exploration. Experience shows that many users prefer to learn how to use a system by exploring its functionality hands on, and not after sufficient training or examination of a user's manual. Therefore, the kinds of checks that are made during the walkthrough ask questions that address this exploratory kind of learning. To do this, the evaluators go through each step in the task and provide a story about why that step is or is not good for a new user.

Pluralistic Usability Walkthroughs. Pluralistic Usability Walkthroughs (Bias, 1994) is a usability evaluation method that brings together representative users and system designers into a design session to discuss new ideas.

The discussion is based on tasks that the participants try to perform with the help of hardcopy panels of the system. The participants get a set of hardcopies of the dialogues that they need to perform the given tasks.

The system designers usually act as "living system documentation" and keep answering questions that are requested by the users. In this way, the users are able to carry on with their tasks and the designers get valuable hints for their system's documentation.

Bias (1994) gives five defining characteristics of the pluralistic usability walkthrough:

1. The method includes three types of participants in the same walkthrough session: users, system designers and usability experts.

2. The system is presented with hardcopy panels and these panels are presented in the same order as they would appear in the system.

3. All participants take the role of a user.

4. The participants write down the actions they would take to perform the given tasks.

5. The group discusses the solutions to which they have reached.

The administrator first presents a correct answer. Then the users describe their solutions, and only after that, do the designers and usability experts offer their opinions. The users in a pluralistic usability walkthrough session are representative users matching the system audience descriptions. The system designers may be architects or coders. A usability expert administers the session. The administrator's role is to make sure the designers' attitude to the users' comments remains positive. If the system designers try to explain away any problems, the users' willingness to give comments soon vanishes.

According to Bias (1994), the pluralistic usability walkthrough method provides reliable data on a particular user interface panel in much the same way as traditional usability testing. The efficiency of the system, the user interface flow and navigation throughout the interface, on the other hand, cannot be reliably evaluated with this method. Compared to usability testing, pluralistic usability walkthrough is better in revealing uncertain decisions. In usability tests, these "lucky guesses" might go unnoticed, but in the pluralistic usability walkthrough sessions, the users can easily report that although they had the right solution, they were not sure about it.

Pluralistic usability walkthroughs are based on scenarios, which promotes the understanding of the user roles. However, they can be slow and limited to a single thread of testing. A fixed sequence of hard-copy panels limits the simulations that users can perform (no browsing or exploring). Also, alternative paths for the same task are not explored. Because the walkthrough is dependent on all users finishing each task before discussion can begin, the session can feel difficult.

Heuristic evaluation. Heuristic evaluation is the most popular of the usability inspection methods. Heuristic evaluation is done as a systematic inspection of a user interface design for usability. The goal of heuristic evaluation is to find the usability problems in the design so that they can be attended to as part of an iterative design process. Heuristic evaluation

involves having a small set of evaluators examine the interface and judge its compliance with recognized usability principles (the "heuristics").

In general, heuristic evaluation is difficult for a single individual to do because one person will never be able to find all the usability problems in an interface. Luckily, experience from many different projects has shown that different people find different usability problems. Therefore, it is possible to improve the effectiveness of the method significantly by involving multiple evaluators. Nielsen (1992) provides perhaps the most well-known list of usability heuristics:

- Visibility of system status: the system should always keep users informed about what is going on, through appropriate feedback within reasonable time.
- Match between system and the real world: the system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.
- User control and freedom: users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.
- Consistency and standards: users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.
- Error prevention: even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.
- Recognition rather than recall: minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.
- Flexibility and efficiency of use: accelerators - unseen by the novice user - may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

- Aesthetic and minimalist design: dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.
- Help users recognize, diagnose, and recover from errors: error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.
- Help and documentation: even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

Collaborative usability inspections. Collaborative usability inspections are a structured way of reviewing an interactive system's level of usability (Constantine and Lockwood, 1999). It is a process involving designers, developers, end users, graphic designers and usability specialists, all engaged in identifying usability defects and user interface inconsistencies.

These inspections can be conducted at almost any phase of development. The usability inspection team can inspect almost any deliverable or development artifact, including other sites or application versions.

Constantine and Lockwood (1999) provide a list of advice in order to maximize the efficiency of this process, as well as to find the maximum possible number of usability defects. The inspection is not aimed at designing, discussing, debating, or congratulating designers or developers but instead to note and record good features that are worth maintaining.

The usability defects that are supposed to be found using collaborative usability inspections should represent evident deviations from well-agreed usability principles like the ones described previously.

There are many methods and approaches for linking every issue described in this section in order to form a solid process methodology. In the next subsection, we will describe two state of the art methods that were the most influential throughout this research: Usage-centered Design (Constantine and Lockwood, 1999) and Wisdom (Nunes, 2001).

2.1.8 *Usage-centered Design*

Usage-centered Design (Constantine and Lockwood, 1999) is a model-driven engineering method essentially based on three simple models: a user model, a task model, and an interface model.

Usage-centered Design employs abstract models that have been tuned through experience and practical, real-world cases in order to capture and clarify the essence of users, tasks, and interfaces in the most expeditious and efficient manner. Regarding users, the interest lies uniquely in the roles they play in relation to a system, and the designer tries to capture the salient and significant aspects of these relationships in the form of an abstract user role model. For a task model, task cases are employed: these are use cases defined by "abstract, generalized, technology-free descriptions" (Constantine and Lockwood, 1999; 2001). For the interface itself, the designer begins with simple models of interface contents and maps of navigation paths or other forms of abstract prototypes (Constantine, 1998) to help achieve the structure and organization right before becoming buried in the details of the real user interface.

By relying on abstract models,

> *"Usage-centered design avoids being seduced and led astray by distracting details. Of course, for everything there is a time, and the time to become immersed in details is after you have explored and fully understood the territory; detailed and local decisions come in their turn once general and global issues have been truly resolved. For holding the devilish details at bay until you are fully equipped to deal with them, the pay off is enormous."*
> (Constantine and Lockwood, 1999)

Abstraction works, leading to genuine innovation and truly world-class results such as the ones described in (Constantine and Lockwood, 2002a; Constantine, 2002; Windl, 2002).

Usage-centered Design consists of three primary models:

- the *User Role Model* – captures the characteristics of the users roles, and the relationship of users with the system; There are many aspects to be covered: the different user roles, the attitude of each role towards the system, the purpose and frequency

of interactions for each role, the volume and direction of data being exchanged between the user and the system;

- the *Task Model* – represents the tasks that users are interested to accomplish with the system, based on the essential use cases which are simplified, abstract, technology-free use cases that represent the user tasks (introduced by Constantine and Lockwood in 1992); they focus on user intentions and system responsibilities and the interrelationships among task cases is shown in a form of a map;
- the *Content Model* – models the content and organization of the user interface needed to support the tasks (rather than UI look & feel): it is abstract in the sense that it is independent of the actual UI look & feel; it models the contents and organization of the UI in abstract terms (no design details given); it defines interaction contexts, which are defined as the contexts within which the user interaction takes place; the content model also serves as a bridge between the task model and the representational prototype.

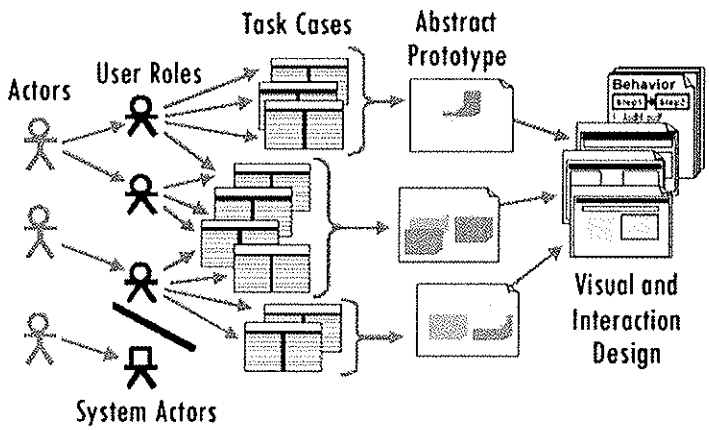


Figure 2.10 A closer look at the Usage-Centered Design Process [taken from (Constantine, 2003)].

Figure 2.10 demonstrates the logical process in Usage Centered Design, from the original recognition of actors to the final UI design.

Unlike other software engineering methods, the UI design is pushed all the way to the final stages of the process in usage centered design. In other words it is part of the solution description, rather than the problem definition.

As shown in the diagram of Figure 2.10, the final step of the process – the visual and interactive design – is directly driven from the abstract prototype.

Usage Centered Design is an "outside-in" design approach, meaning that first and foremost the external user needs will be analyzed, then the external system UI that supports user tasks is designed. The internal system is then developed in order to support this external UI. Therefore, essential use cases are the primary resource, and concrete use cases are written later when required.

2.1.9 *The WISDOM Method.*

The WISDOM method (Whitewater Interactive System Development with Object Models) was originally conceived in order to meet the development needs of small software teams who had to develop and maintain interactive systems following a systematic process (Nunes and Cunha, 2001).

WISDOM is a new, simplified, software development method proposal which is comprised of three important aspects:

- the WISDOM *process*, which defines an alternative to the Unified Process, specifically adapted to software development undertaken by small groups of people in a controlled evolutionary prototyping model;
- the WISDOM *architecture* introduces new UML constructs supporting user profiles modeling, interaction, dialog, and presentation issues modeling;
- the WISDOM *notation* is an extension of the UML which defines a new set of modeling elements that support the WISDOM process and architecture.

Since WISDOM is a method focused on interactive systems, it uses Constantine's essential interpretation of use cases, described earlier. WISDOM is guided by the use cases and the use cases also drive the classes of analysis, task flows, interaction spaces and even non-functional requirements.

Task flows are crucial in the WISDOM approach. A task flow corresponds to a description of the user intentions and the system's responsibilities during the realization of a specific task. That description is accomplished in a technology-independent way.

In the solution proposed in WISDOM, use cases follow a diagrammatic representation that facilitates the integration with artifacts created from participatory sessions and with

UML activity diagrams which are easy to maintain, relate to other modeling elements and expand as the knowledge of the problem increases. In this manner, WISDOM combines essential use case narratives proposed by Constantine with participatory techniques.

This representation, which is coherent with the UML standards, promotes requirements gathering through participatory techniques, since it permits a better manipulation than essential use case narratives. Task flows are initially built in participatory sessions using "low-tech" materials ("post-it" notes) and they are subsequently translated into UML diagrams which are easy to maintain, extend, relate and document by the development team.

The goal is to ensure that the development team will produce a UI capable of reflecting the product's usage structure, and not the application's internal structure (Nunes and Cunha, 2000).

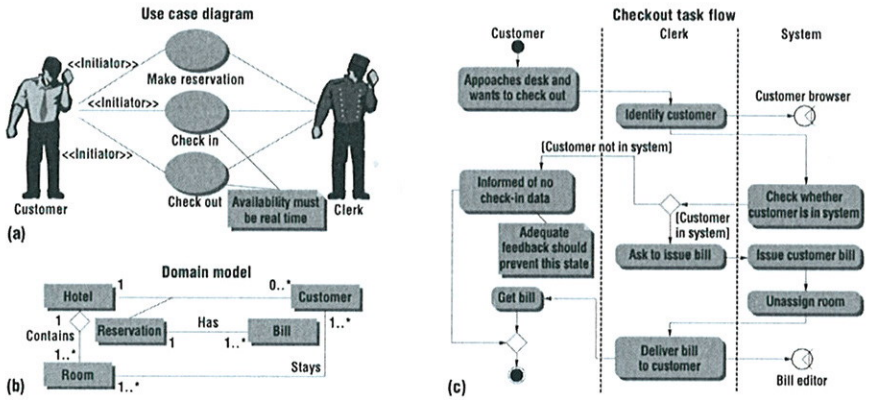


Figure 2.11 An example of a Wisdom model for a hotel reservation system: (a) essential use cases' model for two distinct users; (b) domain model; (c) task flows for the "check-out" use case. [Taken from (Nunes, 2000)].

Figure 2.11 illustrates the use case model and task flows for a classical example: a hotel reservation system (Nunes and Cunha, 2000, 2001), usually stated as follows:

> *"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."*

The essential use cases diagram (a) has an example of incorporating a non-functional requirement ("Availability must be real time"). The domain model (b) represents the most important classes in the context of the system. The task flows (c) are modeled in UML as an activity diagram which corresponds, in Figure 2.11, to the use case "Checkout".

During the analysis phase, the requirements described in the system's architecture of use are refined and structured. The main analysis activities are to identify and structure the analysis-specific classes as well as WISDOM-specific classes (interaction spaces and tasks).

Analysis classes represent abstractions of concepts that were captured in the requirements phase. They are focused on functional requirements, leaving non-functional requirements to the design phase. Following the UML standard, analysis classes are divided in three types: entity (passive, often persistent information), control (complex business logic) and boundary (communication with external entities).

The WISDOM method extends the UML's conceptual analysis model by introducing two new classes that are specifically focused on the interaction with the user. Interaction spaces are class stereotypes that model the interaction between the system and the actors. They're responsible for receiving and presenting information from/to the users. Interaction space classes have actions instead of operations and stereotyped attributes with input and output elements which model information received from and presented to the user, respectively.

The other class is Task. Task classes are responsible for task sequencing and for the consistency in the presentation entities (interaction spaces). Tasks encapsulate the complex temporal dependencies as well as other restrictions among activities, thus isolating changes in the dialog structure.

Figure 2.12 shows the notations employed for the interaction and analysis model in WISDOM.

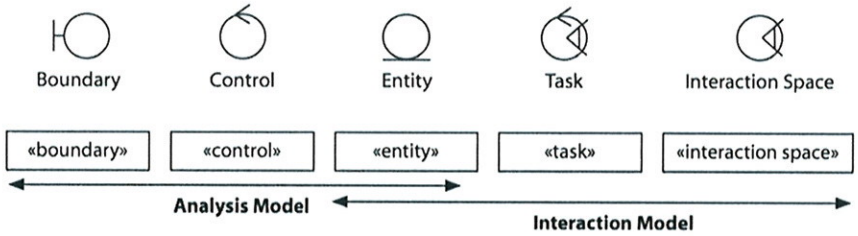


Figure 2.12 Alternative notations for the interaction and analysis model's class stereotypes [Taken from (Nunes, 2001)].

Figure 2.13 illustrates an example of an architectural model in WISDOM for the same hotel reservation system described previously, showing the analysis classes on the right, and the interaction classes on the left.

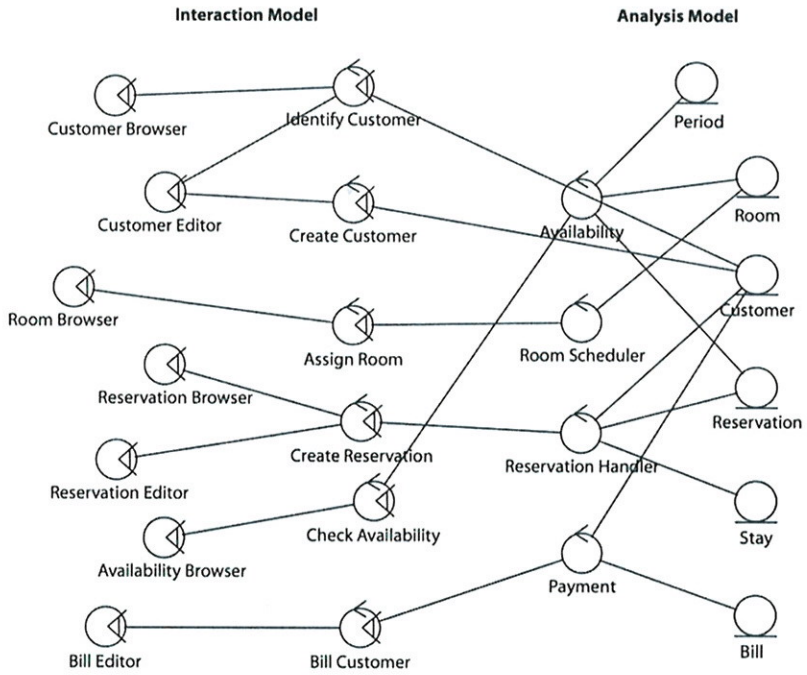


Figure 2.13 An example of a Wisdom architecture interaction and analysis model, for a hotel reservation system (Nunes and Cunha, 2000).

In order to support the modeling of UI presentation issues during the design phase, the WISDOM method proposes the following extensions to the UML (Nunes and Cunha, 2000):

- «Interaction Space», a class stereotype that represents the space contained in the UI where the user interacts with all tools and materials during the course of a task or set of inter-related tasks;
- «navigates», an association stereotype between two interaction spaces that denotes a change from one interaction space to another, a change which is triggered by the user;
- «contains», an association stereotype also between two interaction space classes which denotes that the source class (the container) contains the target class (the contained); this association stereotype can only be used between two interaction space classes and is unidirectional;
- «input element», an attribute stereotype that denotes information received from the user, i.e., information upon which the user can operate/manipulate;
- «output element», an attribute stereotype which denotes information presented to the user, i.e., information that the user receives but does not manipulate;
- «action», an operation stereotype that denotes something the user can perform in the UI which will cause a significant change in the internal state of the system.

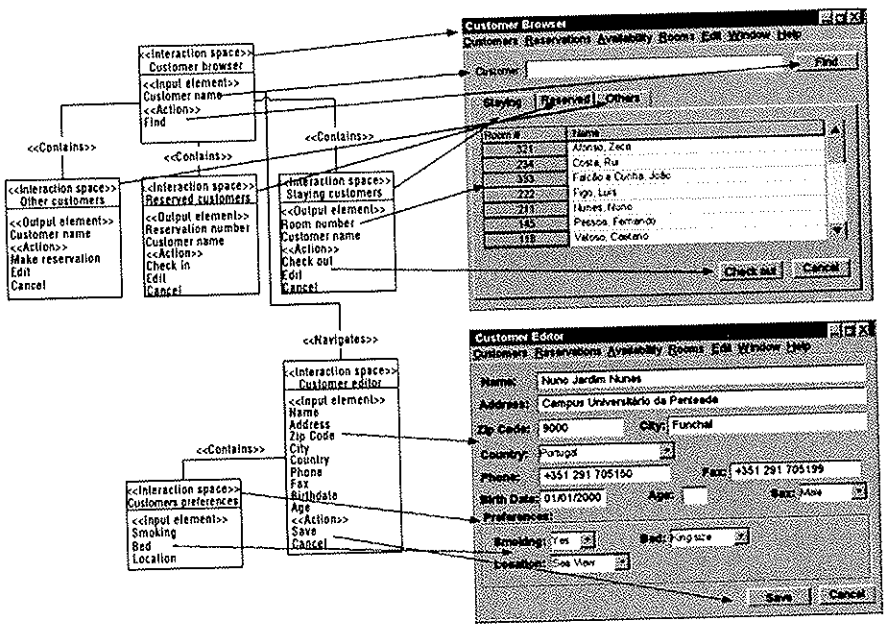


Figure 2.14 Transformation of the Wisdom presentation model from the hotel reservations system into a concrete UI [taken from (Nunes and Cunha, 2000)].

Figure 2.14 shows an example of how a WISDOM presentation model can be transformed (using e.g. XSLT stylesheets) into a concrete UI (Nunes and Cunha, 2000). In this example (the hotel reservation system), there are five detailed interaction spaces (on the left). The resulting prototyped user interface is on the right. Dependencies from left to right show design decisions the UI technology constrains. Several different mappings of WISDOM interaction spaces exist. For example, the containment relationship between the customer browser interaction space and the three types of customer interaction spaces maps as a tabbed GUI element. Also, as illustrated in the figure, input elements map as GUI elements that the user is able to manipulate (editable text fields, combo boxes, and so on) and output elements as GUI elements that the user cannot manipulate (non-editable fields, icons, and so on). Actions (and navigational relationships, sometimes) map as buttons.

WISDOM was implemented in several small software companies in Portugal (Nunes and Cunha, 2000). Experience showed that small companies want to improve their practices and can take advantage of new developments in software engineering. Although WISDOM was tailored for small companies, small development teams within large development companies or user organizations can also use it. WISDOM has been efficient in a wide range of projects, such as Web site designs, interactive web applications, decision support systems, and distributed embedded systems.

2.1.10 Using Patterns for UI Design

In the mid-90's, the practice of software architecture was deeply changed by the publication of Gamma et al. (1995), a collection of patterns describing object-oriented "micro-architectures". Gamma et al. describe patterns for managing object creation, composing objects into larger structures, and coordinating control flow between objects. Their book provides numerous examples where using composition rather than inheritance can improve the reusability and flexibility of code.

Patterns for software engineering were born in the work of Alexander architectural patterns (Alexander et al., 1977). He defined and coined the term pattern as it is employed today. Although his texts were around architectural issues, the ideas presented can be ap-

plied to many other disciplines. In particular, to software engineering and interaction design.

Alexander argues that current architectural methods lead to projects that do not satisfy neither the real needs of users, neither the requirements: in other words, they don't improve human condition, which is ultimately the final goal in any design or engineering effort.

In the book *The Timeless Way of Building*, Alexander (1979) proposes to capture timeless design ideas in order to improve the human level of comfort. His proposed paradigm is built on three key concepts:

- The Quality without a name: the essence of useful things that brings those things qualities like freedom, completeness, comfort, durability, resistance. It's what makes us feel satisfied about a product or a design.
- The Gate: this is the mechanism through which it is possible to achieve the Quality. It's a common pattern language that allows the creation of different designs to satisfy multiple needs. The Gate leads to the Quality.
- The Timeless Way. Using the Way, the patterns defined in the Gate can be applied and progressively combined in order to produce designs that achieve the Quality.

There are many formats for specifying patterns. According to Alexander's format, these are the sections that should be completed in order to describe a given pattern:

Name. The name should be short and meaningful.

Problem. A phrase or paragraph describing the goal of the pattern, i.e., the problem the pattern proposes to solve.

Context. The preconditions upon which the problem and its solution are recurrent and because of which preconditions is the proposed solution a desirable one. The context informs about the applicability of the pattern, it can be regarded as the system's initial configuration before the application of the pattern.

Forces. A description of the relevant forces and restrictions and how these interact or conflict. Forces reveal the complexity of the problem and define the types of interchange

that should be considered in the presence of tensions among forces. A good pattern description should reflect all the forces that have an impact over the pattern.

Solution. Static relationships and dynamic rules that describe how to achieve the desired result. Solutions should contain a series of steps that allow the construction of the desired product.

Examples. One or several examples of application of the pattern that are capable of illustrating: an initial context, how to apply the pattern in that context, and how that application transforms the context. Examples are crucial to the problem's understanding and the application of the pattern.

Rationale. An explanation justifying the steps or rules described by the pattern. The rationale is an explanation for the usefulness of the pattern: it explains how it works, why it works and why it is a good solution.

Related patterns. Some patterns might be inter-related, sharing the same forces and exhibiting similar solutions.

Known uses. Known occurrences of the pattern help to justify and validate a pattern by effectively showing that it is a common solution to a recurring problem.

At its simplest, a set of patterns is a repository of best practices, expressed at a level of abstraction that is more general than bits on a screen, but more specific than heuristics or principles.

In her book *Designing Interfaces*, Jennifer Tidwell (2005) captures the UI design best practices as design patterns: solutions to common design problems, tailored to the situation at hand. Each pattern contains practical advice that one can put to use immediately, plus a variety of examples. In order to illustrate the usefulness of this compiled collection of UI knowledge, we will briefly describe two examples concerning three different UI design problems: showing complex data and getting input from users. The whole collection is available online at `http://time-tripper.com/uipatterns`.

Sortable Table. One pattern that facilitates the display of complex data is the sortable table pattern (Tidwell, 2005) should be used when the UI displays multivariate information that the user may want to explore, reorder, customize or search for a single item.

First, it facilitates exploration. A user can now learn things from the data that they may never have been able to see otherwise – how many of this kind? what proportion of this to that? is there only one of these? What's first or last? etc. Suddenly, it becomes easier to find specific items, too; a user need only remember one attribute of the item in question (e.g. its last-edited date).

As explained by Tidwell when describing the example of Figure 2.15, "choose the columns (i.e., the variables) carefully. What would a user want to sort by or search for? Conversely, what doesn't need to be shown in this table – what can be hidden until the user asks for more detail about a specific item?"

The table headers (see Figure 2.15) should have some visual affordance that they can be clicked on. Most have beveled, button-like borders. Up-or-down arrows should be used to show whether the sort is in ascending or descending order, and the presence of an arrow shows which column was last sorted on – a fortuitous side effect! Designers should also consider using Rollover Effects on the headers to reinforce the impression of clickability.

One should try to use a stable sort algorithm. What this means is that if a user sorts first by name, then by date, the resulting list will show ordered groups of same-date items that are each sorted by name within the group. In other words, the current sort order will be retained in the next sort, to the extent possible. Subtle, but very useful.

Name	Size	Type	Modified
demo		File Folder	8/12/2001 8:38 PM
doc		File Folder	8/12/2001 8:38 PM
frameworks		File Folder	8/12/2001 8:38 PM
javadoc		File Folder	8/12/2001 8:38 PM
lib		File Folder	8/12/2001 8:38 PM
index.html	1 KB	HTML Document	5/1/2001 12:03 PM
license.html	14 KB	HTML Document	5/1/2001 12:04 PM
release_notes.html	1 KB	HTML Document	5/1/2001 12:03 PM
rn_connect.html	1 KB	HTML Document	5/1/2001 12:03 PM
rn_dev.html	25 KB	HTML Document	5/2/2001 4:53 PM
rn_sync.html	1 KB	HTML Document	5/1/2001 12:03 PM

Figure 2.15 Sortable table example, taken from MS Windows Explorer.

Input hints. Input hints should be used when the UI presents a text field, such as the example in Figure 2.16, but the kind of input it requires isn't obviously understandable to all users. By writing a short example or explanatory sentence, and putting it below the text field the user gets feedback on the input's format. Two examples conjoined by "or" works

fine as well. The text should be kept small and inconspicuous, though readable; consider using a font two points smaller than the label font. (One point's difference will look more like a mistake than an intended font-size change!)

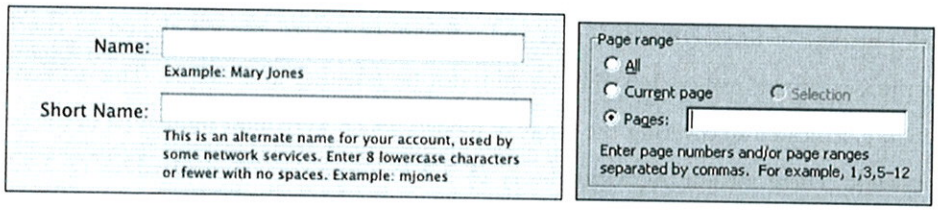


Figure 2.16 Input hints examples.

While some of the patterns are very dialog-oriented, like smart-selection, rollover effects, or edit-in-place, some patterns are also extremely visual: deep background, for instance, is an example of a pattern for making web pages look more distinctive or aligned with a branding strategy. The solution is to use a background that has the following characteristics: a soft focus (keeping lines fuzzy), color gradients, depth cues and no strong focal points (in order not to compete with the content for the user's attention.

2.2 Models for Human Work

A central theme of this thesis is the elaboration of models that capture and describe the work practices of interaction designers and software engineers. Models should be concise descriptions of the reality, but they should also be essential and simple. This section is therefore dedicated to covering the range of models which were most influential in the present work. These include Activity Theory (§ 2.2.1), Cognitive Work Analysis (§ 2.2.2), Traeetteberg's Cube (§ 2.2.3), Wu and Graham's Workstyle Model for Software Design (§ 2.2.4), and the recently formed Human Work Interaction Design field (§ 2.2.5).

2.2.1 Activity Theory

The field of Human-Computer Interaction has been dominated by a cognitive perspective in its approach to research. In the cognitive perspective, people and computers are analyzed as equal members of an information processing system. Inputs and outputs of information are exchanged between the two as the individual works to perform a specific task with the computer (e.g. creating a graphical chart). Early work in HCI concentrated solely on this relationship between the human user and computer, ignoring the larger contextual factors that influence computer use.

For example, the user who wants to create a graphical chart is most likely not working in an isolated room, moving step-by-step through each of the procedures necessary to make the chart until it is complete. Rather, while working to make the chart, the user is interrupted by co-workers, reads instructions about graphical charts from a book, and performs other tasks like answering e-mail in-between all of these other activities.

Activity Theory was developed by the Russian psychologists Vygotsky, Leont'ev and others, (see (Vygotsky, 1978; Leont'ev, 1978)) with work beginning in the 1920's. There is a thriving Activity Theory tradition in HCI studies in Scandinavia and increasing interest in Activity Theory in other European countries, the U.S., Canada and Australia, as well as continuing work in Russia.

Context is essential in getting at an understanding of computer use because it influences the nature and outcome of the task at hand. Nardi (1996) presents activity theory as a framework and perspective to be used by HCI researchers because it accommodates con-

text. Activity theory is not a predictive framework, rather a way to organize rich descriptions of individual activity. Activity theorists seek to analyze "individual consciousness" by looking at how the user achieves their goal in relation to the particular tools and situation. Individual consciousness manifests itself through the actions of the individual, embodied through real world practices.

Activity Theory is actually comprised of a set of basic principles that constitute a general conceptual system, rather than a highly predictive theory. The basic principles of Activity Theory include the hierarchical structure of activity, object-orientedness, internalization/externalization, tool mediation, and development.

Hierarchical structure of activity. In Activity Theory the unit of analysis is an activity directed at an object which motivates activity, giving it a specific direction. Activities are composed of goal-directed actions that must be undertaken to fulfill the object. Actions are conscious, and different actions may be undertaken to meet the same goal. Actions are implemented through automatic operations. Operations do not have their own goals; rather they provide an adjustment of actions to current situations. Activity Theory holds that the constituents of activity are not fixed, but can dynamically change as conditions change.

Object-orientedness. The principle of "object-orientedness" (not to be confused with object-oriented programming) states that human beings live in a reality that is objective in a broad sense: the things that constitute this reality have not only the properties that are considered objective according to natural sciences but socially/culturally defined properties as well.

Internalization/externalization. Activity Theory differentiates between internal and external activities. It emphasizes that internal activities cannot be understood if they are analyzed separately from external activities, because they transform into each other. Internalization is the transformation of external activities into internal ones. Internalization provides a means for people to try potential interactions with reality without performing actual manipulation with real objects (mental simulations, imaginings, considering alternative plans, etc.). Externalization transforms internal activities into external ones. Externalization is often necessary when an internalized action needs to be "repaired", or scaled. It is

also important when a collaboration between several people requires their activities to be performed externally in order to be coordinated.

Mediation. Activity Theory stresses that human activity is mediated by tools in a broad sense. Tools are created and transformed during the development of the activity itself and carry with them a particular culture. So, the use of tools is an accumulation and transmission of social knowledge. Tool use influences the nature of external behavior and also the mental functioning of individuals.

Development. In Activity Theory, development is not only an object of study, it is also a general research methodology. The basic research method in Activity Theory is not based on traditional laboratory experiments but the formative experiment which combines active participation with monitoring of the developmental changes of the study participants. Ethnographic methods that track the history and development of a practice have also become important in recent work.

Integration of the principles. These basic principles of Activity Theory should be considered as an integrated system, because they are associated with various aspects of the whole activity. A systematic application of any of these principles makes it eventually necessary to engage all the other ones.

In contrast to cognitivism, humans and computers are never equal entities in activity theory. The computer is viewed as a tool that mediates human activities for a larger purpose beyond just interacting with the computer. For a user who wants to make a graphical chart, activity theory would view the user's purpose (or "object" as it is called in activity theory) to be something like, "finalize the report" or "learn how to use Excel". Once the user's object is defined in this way, it is apparent that the activities of the user will be interpreted quite differently depending on the object.

Activity theory is similar on the surface to the work of Suchman's (1987) situated action. Situated action analyzes user behavior through the emergent moment-by-moment actions of users during a particular activity. Both perspectives are focused on this meaningful human action in context, however, there is a key difference. Situated action dismisses the role of pre-determined intentions and goals of the user as part of the analysis. Suchman believes that a post hoc rationalization for intention can be described after the outcome,

but this rationalization in no way influences how the activity unfolds. There is no intentionality in situated action, what happens is always developing ad hoc out of the situation.

Activity theory, on the other hand, gives primacy to the goal (object) as a key element in the analysis. Nardi (1996) gives an example where if we're analyzing a videotape of two people looking at the sky, one who is a birdwatcher and the other a meteorologist, without knowing their intentions it is unclear how to interpret why they look upwards. However, it can be argued that if Suchman defined the activity not as "looking upward," but rather as "bird watching" and "observing weather" that the issue of plans and intentionality would no longer be a problem for the analysis.

Another area of difference in which Nardi favors activity theory over situated action is that of persistent structures. Nardi defines persistent structures as artifacts, institutions, and cultural values that "stretch across situations and activities that cannot be properly described as simply an aspect of a particular situation". In activity theory, an artifact like a computer spreadsheet is analyzed as having a particular set of design features that will remain persistent across activities. These design features can be identified as a structure that shapes how the user's activity unfolds.

Parts of activity theory and situated action are very useful in informing some research development. However, activity theory has its limitations in that its focus is only on the individual, and does not purport to account for the activities of a group (see Engeström (1987) for attempts at extending activity theory to a group level). Another issue with activity theory is that it cannot account for multi-layered consciousness: i.e. the activities of the user are not single and linear: the user in a context can be both listening to other people and checking e-mail at the same time.

2.2.2 *Cognitive Work Analysis*

Cognitive Work Analysis (Vicente, 1999) is a work-centered conceptual framework developed by Rasmussen et al. (1994) to analyze cognitive work. The purpose of Cognitive Work Analysis (CWA) is to guide the design of technology for use in the work place. It is unique because of its ability to analyze real-life phenomena while retaining the complexity inherent in them. When applied to information behavior, the approach guides the analysis of human-information interaction in order to inform the design of information systems.

CWA's theoretical roots are in General Systems Thinking (Weinberg, 1975), Adaptive Control Systems (Tao and Kokotovic, 1996), and Gibson's Ecological Psychology (Gibson, 1986), and it is the result of the generalization of experiences from field studies which led to the design of support systems for a variety of modern work domains, such as process plants and libraries. In the context of Information Science, the concept "information system" refers to any system, whether intellectual or computerized, that facilitates and supports human-information interaction.

Unlike the common approach to the design of information systems - design and development first and evaluation later - CWA evaluates first the system already in place, and then develops recommendations for design. The evaluation is based on the analysis of information behavior in context. CWA has been successfully applied to the evaluation and design of information systems. For example, it guided the development of the first retrieval system for fiction called BookHouse (Pejtersen 1989; Rasmussen et al., 1994; Pejtersen, 1992). Based on the analysis of reference interviews in public and school libraries, Pejtersen developed a fiction retrieval system, with a graphical user interface, in which users can look for books by a variety of attributes, such as the subject, historical period, mood, and the cover design. It serves children and adults, as well as library catalogers. The system also caters to various strategies: users can just browse without any particular attribute in mind, look for a specific book, or look for books that are similar to one they liked. CWA was used to analyze data collected in a study of Web searching by high school students (Fidel et al., 1999). In this study, the framework proved to be very powerful in helping to uncover the problems that students experienced when using the Web to search for information, and offered recommendations for designs that can alleviate such problems. Pejtersen and her colleagues also completed the COLLATE project that supports multi-institutional collaboration in indexing and retrieval among the national film archives of Germany, Austria, and the Czech Republic (Albrechtsen et al., 2002, Hertzum et al., 2002).

Cognitive Work Analysis considers people who interact with information "actors" involved in their work-related actions, rather than as "users" of systems. Focusing on information behavior on the job, CWA views human-information interaction in the context of human work activities. It assumes that in order to be able to design systems that work harmoniously with humans, one has to understand:

- The *work* actors do,
- Their information *behavior*,
- The *context* in which they work, and
- The *reasons* for their actions

Therefore, CWA focuses simultaneously on the task the actors perform, the environment in which it is carried out, and the perceptual, cognitive, and ergonomic attributes of the people who do the task. A graphic presentation of the framework is given in Figure 2.17. In this presentation each set of attributes mentioned above is designated with a circle and is considered a dimension for analysis. Thus, each dimension is a host of attributes, factors, or variables - depending on the purpose and method of a study. In addition to the dimensions for analysis, CWA provides several templates to support both analysis and modeling. These templates are particularly suitable for the analysis of complex and dynamic phenomena.

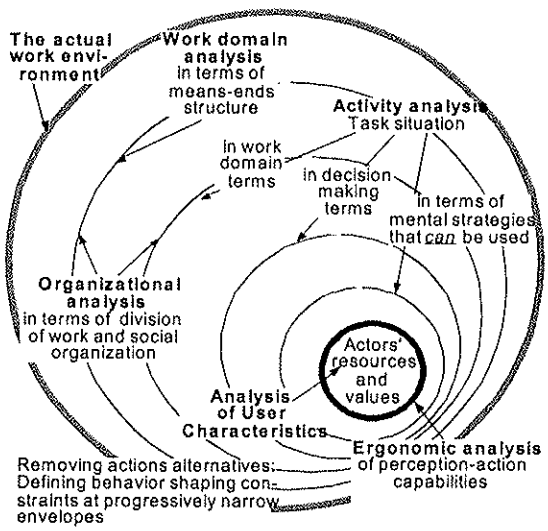


Figure 2.17 The dimensions of Cognitive Work Analysis [taken from (Pejtersen, 1989)].

Table 2.1 shows a few examples of questions one may want to ask when analyzing along each dimension.

Dimension	Questions that can be asked during Analysis
Environment	What elements outside the organization affect it?
Work Domain	What are the goals of the work domain? The constraints? The priorities? The functions? What physical processes take place? What tools are employed?
Organizational Analysis	How is work divided among teams? What criteria are used? What is the nature of the organization, hierarchical, democratic, chaotic? What are the organizational values?
Task Analysis in Work Domain Terms	What is the task (e.g., design of navigation functionality)? What are the goals of the task that generated an information problem? Constraints? The functions involved? The tools used?
Task Analysis in Decision Making Terms	What decisions are made (e.g., what model to select for the navigation)? What information is required? What sources are useful?
Task Analysis in terms of strategies that can be used	What strategies are possible (e.g., browsing, the analytical strategy)? What strategies does the actor prefer? What type of information is needed? What information sources does the actor prefer?
Actor's Resources and Values	What is the formal training of the actor? Area of expertise? Experience with the subject domain and the work domain? Personal priorities? Personal values?

Table 2.1 Questions that can be asked when performing Cognitive Work Analysis.

Although the dimensions are laid out in a certain order, employing them in actual projects follows no fixed sequence. Because of the interdependence among the dimensions, a researcher moves from one dimension to another in an iterative process. The path of this movement is determined by the particular problem at hand, and also by pragmatic considerations.

2.2.3 *Traeetteberg's Cube*

Traeetteberg (2003) claims that current UCD tools should be able to cover a three-dimensional cube of Perspective (moving from problem/requirements space to the solution/design), Granularity (from high level to low level) and Formality (formal, i.e. machine-understandable versus informal i.e. context-dependent). It is suggested that in the course of designing an interface, several languages have to be used to cover all the needed perspectives (see Figure 2.18).

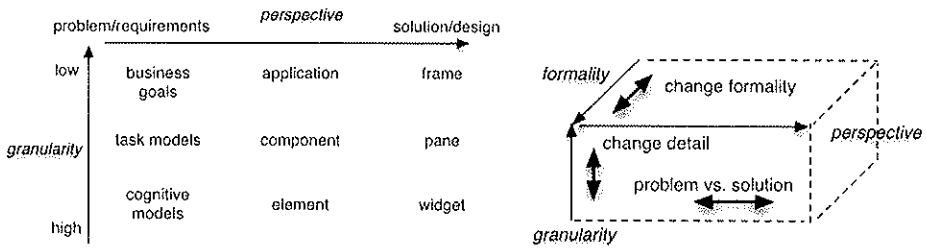


Figure 2.18 Left: an interpretation of the granularity versus perspective. Right: movements in the representation framework of Traeetteberg (2003).

As an example, Traeetteberg (2003) suggests one interpretation of the granularity level across different perspectives (see the left side of Figure 2.18). A task is performed to achieve a goal, and is often supported by a specific component composed of dialogue elements, which is placed in a pane containing widgets.

2.2.4 *Workstyle Models for General Software Development*

Software design is often a team activity and most projects involve stakeholders with different backgrounds that must cooperate in many different and interrelated activities.

Wu and Graham (2004) describe a novel model for recording the working style of people using an interactive system. Workstyle modeling complements task modeling by providing information on how people communicate and coordinate their activities, and by showing what style of artifact is produced.

The workstyle model was developed in the context of the Software Design Board project, a project aiming to provide better tools for software design. The model was validated through evaluation of existing design tools, and motivated the design of a new software design tool. It is comprised of eight axes: four of them describe collaboration style (Location, Synchronicity, Group Size and Coordination); the remaining four describe the nature of the artifact being produced (Syntactic Correctness, Semantic Correctness, Archivability and Modifiability).

The workstyle model for software design has the advantage of being simple to apply and clearly showing where a tool can fail to match the intended work context. However, it is not sufficient for capturing UI specific activities.

2.2.5 *Human Work Interaction Design*

Human Work Interaction Design is a recent research area which was born under the auspices of IFIP's Working Group 13.6. In a recent workshop (Clemmensen et al., 2005), new themes and directions of research on human work analysis and design to support it have been outlined. The main targets of the work group are the analysis of the variety of complex work and life contexts found in different businesses, and design methods for supporting them. In a subsequent initiative, a new Working Conference (HWID'06) was launched and a book arising from the conference was published (Clemmensen et al., 2006) describing several interesting case studies about improving collaboration in software development activities, representing complex information structures in high-volume manufacturing, designing resource allocation planning systems, or design sketching for space and time applied to air traffic control systems.

The aims of the HWID working group were recently defined in their official IFIP web site as follows:

- To encourage empirical studies and conceptualizations of the interaction among humans, their different social contexts and the technology they use both within and across these contexts;
- Promote the use of knowledge, concepts, methods and techniques that enable user studies to achieve a better apprehension of the complex interplay between individual, social and organizational contexts, and therefore achieve a better understanding of *how* and *why* people work in the ways they do;
- Promote a better understanding of the relationship between work-domain based empirical studies and iterative design of prototypes and new technologies;
- Establish a network of researchers, practitioners and domain/subject matter experts working within this field.

Thus, on an overall level the working group aims at establishing relationships between extensive empirical work-domain studies and HCI design.

It is a challenge to design applications that support users of technology in complex and emergent organizational and work contexts, which is why the working group strongly believes that plenty of opportunities exist to focus on methods, theories, tools, techniques and prototype design on an experimental basis.

Under these circumstances, the primary question is less whether we choose to study the use of a particular computer application or, instead, prefer to conduct bottom up empirical experiments of work contexts. The new problem is how we can understand, conceptualize and design for the complex and emergent contexts in which human life and work are now involved. This problem calls for cross disciplinary, empirical and theoretical approaches that focus on Human-Work Interaction Design, meaning that the technology itself – and particularly the design and use of technologies – mediates the interaction between humans and specific work contexts.

2.3 Models and Methods for Evaluating Tools

Our research strategy has primed for an empirically-sound framework that could be used to inform as well as to validate the design of better design tools. This research objective is not new, but the question is very timely: the argument that the practitioner's behavior must be studied in order to make better tools has been used recently by (Seffah and Kline, 2002; Ko and Myers, 2005), just to name a few. In this section, some models and methods aimed at evaluating tools are described and discussed.

2.3.1 The Technology Acceptance Model

Complex work activities increase the difficulty of predicting the level of acceptance of novel technology and how it will be used in practice. An important and open research question is how to translate usability evaluation results into concrete design decisions (Morris and Dillon, 1997; Wright et al. 2000).

The Technology Acceptance Model (TAM) developed by Davis and colleagues (Davis, 1989) is a widely used theoretical model in the Management Information Systems (MIS) field. Basically, it attempts to predict and explain computer-usage behavior, offering both researchers and practitioners a direct, pragmatic instrument to measure a technology degree of acceptance. Morris and Dillon (1997) pointed out that TAM offers HCI professionals a "theoretically grounded approach to the study of software acceptability that can be directly coupled to usability evaluations".

In TAM, shown in Figure 2.19, there are five primary constructs: Perceived Usefulness (PU), the extent to which the user expects the system to improve his/her job performance within an organizational setting; Perceived Ease Of Use (PEOU), the degree to which the user believes the system's use will be free of effort; Attitude toward Using (A), the user's desire to use or favorableness feelings towards using the system; Behavioral Intentions to Use (BI), which measures the strength of a user's intention to use the system in the future; and the Actual Use, i.e. the amount of usage per time unit.

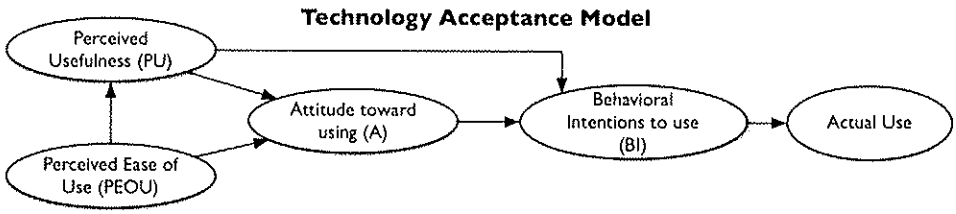


Figure 2.19 Davis's Technology Acceptance Model. It suggests that a person is more likely to actually use a technology if he believes that it will be both useful and usable.

As depicted in Figure 2.19, the actual use of a system is a direct function of the behavioral intentions to use it. These are in turn influenced by perceived usefulness and attitude toward using. Perceived ease of use and perceived usefulness are both crucial to determine the attitude toward using the system.

TAM has been effectively used (Taylor and Todd, 1995; Mathieson, 1991) to predict system's acceptability, but it cannot be used to explain specific design flaws (Morris and Dillon, 1989). However, it presents the important advantage of being a reliable and cost effective way to evaluate systems at any life cycle time.

There has been much research following the lines of the TAM model. Wright et al. (2000) discuss and provide methods for function allocation that take into account work practices in complex work domains. Their focus is not only on work practices but especially on representations, in particular on making work practices representations that can inform early design decisions. They extended a similar line of thought described by (Suchman, 1995) when arguing for making work practices visible.

The Method Evaluation Model (MEM) combines Davis' TAM with Rescher's Theory of Pragmatic Justification (Rescher, 1973), a theory for validating methodological knowledge. This model is aimed at evaluating Information Systems (IS) methods, rather than general systems as in TAM. The central constructs of MEM are Perceived Ease of Use, Perceived Usefulness and Intention to Use. MEM's additional constructs are Actual Efficiency (effort required to apply a method) and Actual Effectiveness (the degree to which a method achieves its' objectives). MEM has been successfully applied, for instance, to functional size measurement of object-oriented web applications (Abrahão, 2005).

2.3.2 *Empirical Studies*

Ko and Myers (2005) propose a framework for studying the causes of software errors in programming systems. It is based on studies of programming activities as well as general research results on human error mechanisms. Their framework is particularly interesting, since it directly inspired the design of new tools and interfaces, an approach philosophy we also pursued. The experimental procedure is very difficult and somewhat intrusive, though. Users were videotaped and asked to think aloud about their actions. The experimenters later studied the recordings in order to uncover fundamental activities in the programmer's work. This resulted in accurate results at the expense of hard, extensive human analysis.

Wu et al. (2003) present a study of collaboration design at a large software company. Their study revealed that designers communicate frequently using a wide variety of communication and collaboration modalities. General-purpose tools are preferred instead of domain-specific applications. There is a clear indication that the flexibility in collaboration is more important to designers than the advantages of domain-specific tools, such as syntax checking of designs or code generation. The preference for informal media (such as paper, post-it notes and whiteboards) found in this study is consistent with observations from other researchers (Landay, 2001; Wagner, 1990).

The study also showed that designers frequently change their physical location throughout the day, in order to support communication. They also change the ways in which they communicate, changing their modalities and styles. Where co-located interaction was possible, there was a great preference for and use of face-to-face communication, often in conjunction with whiteboards.

After this study, the resulting impact on tool design was discussed. In particular, they note that a tool supporting only asynchronous communication, via e-mail or document repositories does not address the predominantly synchronous interactions in which designers engage (Wu et al., 2003).

Newman and Landay (2000) performed a field study about web design practices and they observed that designers employ multiple representations of web sites as they progress through the design process. Among the several work practice studies they refer, Sumner and Stolze (1997) study of speech application designers and Bellotti and Rogers (1997) study of editorial staff at several publishing companies both showed that designers and

editors use multiple intermediate representations of products during their elaboration, something they also noted in their web site design practices study.

Newman and Landay (2000) interviewed eleven designers involved in web site design processes, collected and studied many of the artifacts produced (this included sketches, storyboards, written documents, prototypes, presentations and finished web sites), and after the field interviews they modeled the relationships among the different areas of design, which are shown in Figure 2.20.

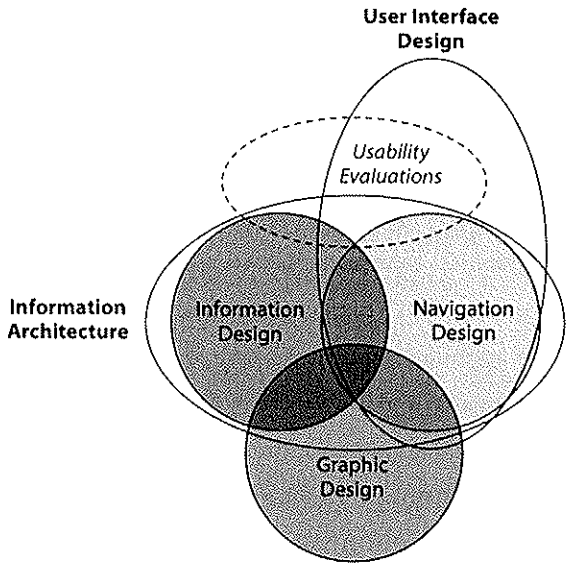


Figure 2.20 The different representations within web site design found by (Newman and Landay, 2000).

From the study's results, they elaborated a series of desirable features for new design tools. These include: the use of an informal User Interface; focus on early design phases; integration with other tools; manage history and variations; and support for multiple representations. A design tool supporting multiple representations "would be an improvement over the current state of the art, in which representations are created using a separate, poorly integrated tools" (Newman and Landay, 2000).

Another way to measure a user's satisfaction is through the SUMI (Software Usability Measurement Inventory) questionnaire (Kirakowski and Corbett, 1993). SUMI is a rigorously tested and proven method of measuring software quality from the end user's point of

view. SUMI is a consistent method for assessing the quality of use of a software product or prototype, and can assist with the detection of usability flaws before a product is shipped.

SUMI is recommended to any organization which wishes to measure and evaluate the usability of software.

SUMI is used specifically to:

- assess the usability and user satisfaction of new products during the test stage;
- make comparisons between products or versions of products;
- set usability targets for future applications;
- highlight good and bad dimensions of an interface.

It works the following way: a number of users (at least 10) from a specific group of users should fill out the SUMI questionnaire. Based on the given answers, and using statistical principals, scores are calculated for efficiency, control, affect, helpfulness and learnability. SUMI provides a graphical comparison between the assessed software and the general usability scores of software in the IT market.

Among some of the advantages for using SUMI is the fact that it is the only available questionnaire for the evaluation and assessment of the usability of software which has been developed, validated and standardized on a European wide basis. SUMI questionnaires are available in, English, French, German, Dutch and a number of other languages.

Product testing and evaluation with SUMI provides a clear and objective measurement of the users' view of the suitability of software for their tasks.

2.3.3 *Cognitive Dimensions of Notations*

Cognitive Dimensions (Green and Petre, 1996) have been proposed both as an evaluation technique for visual programming environments and as a discussion tool for designers. This technique concentrates on the activities rather than the finished product. Based on the observation that HCI creators don't adopt expensive or time-consuming evaluation methodologies, Green proposed what he calls "a framework of user-centered tools".

The framework has evolved over the years and has been proven as a useful set of discussion tools. The dimensions proposed are the following [taken from (Green and Petre, 1996)]:

- *Abstraction Gradient*: what are the minimum and maximum levels of abstraction? Can fragments be encapsulated?
- *Closeness of mapping*: what 'programming games' need to be learned?
- *Consistency*: when some of the language has been learnt, how much of the rest can be inferred?
- *Diffuseness*: how many symbols or graphic entities are required to express a meaning?
- *Error-proneness*: does the design of the notation induce 'careless mistakes'?
- *Hard mental operations*: are there places where the user needs to resort to fingers or penciled annotation to keep track of what is happening?
- *Hidden dependencies*: is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?
- *Premature commitment*: do programmers have to make decisions before they have the information they need?
- *Progressive evaluation*: can a partially complete program be executed to obtain feedback on "How am I doing"?
- *Role-expressiveness*: can the reader see how each component of a program relates to the whole?
- *Secondary notation*: can programmers use layout, color, or other cues to convey extra meaning, beyond the 'official' semantics of the language?
- *Viscosity*: how much effort is required to perform a single change?
- *Visibility*: is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to juxtapose any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?

The Cognitive Dimensions' guidelines have seen a great success both at an academic and industrial level. The guidelines were taken and taken by Steven Clarke and the Visual Studio usability group at Microsoft, who adopted them as a usability inspection technique (similar in spirit to heuristic evaluation (Nielsen, 1992)) and as a language for describing laboratory evaluation results of the Visual Studio .NET IDE, the C# language, and the .NET APIs (Clarke, 2001).

Clarke (2001) is the first publication by a product group taking a user-centered approach to designing programming languages and tools. Their work begins by using scenarios of small programming tasks to drive design. Then, the relevant IDE, library, or language feature are designed. After design, the feature is evaluated in a usability test.

Not every building block in the system undergoes this precise treatment; only aspects that are regarded as important to Microsoft. Klemmer (2004) visited this product group at Microsoft to observe their usability labs and discuss their methods. Each feature "is evaluated with approximately ten developers. Each developer participates in two sessions that are roughly two hours long; the two sessions are about a week apart. The first session is used to ascertain how usable a language/IDE feature is the first time a developer sees it. The second session ascertains how easily the developer retains that knowledge, and how use changes with familiarity" (Klemmer, 2004).

Klemmer (2004) reports that the evaluation work taken by Clarke's approach at Microsoft was similar to his evaluation of the Papier-Mache tool (Klemmer et al., 2004), a toolkit for supporting tangible UI input, except that the comfortable budget at Microsoft allowed a more thorough and extensive application of the methods.

2.4 Conclusions

This chapter presented a brief survey on the usability principles, models, methods and tools that should support modern development of interactive systems. We focused the analysis on the work-capturing models that were most inspirational to the models and tools we will present in the following chapters.

The importance of usability is proven and accepted nowadays. However, more research should be devoted to bringing usability and usefulness to the interaction and user-centered design tools themselves. Software engineers and interaction designers are the primary users of design tools, and one cannot expect them to create usable tools if their own design and development environments are not adequate to their work practices.

In the following chapter, we will survey some of the tools that are currently employed by software engineers and interaction designers in their quotidian endeavors.

3 State of the Art

Interaction Design Tools

> *"Nothing will change unless we can influence the software developers. Even if the programmers agree that the user should be better treated – and they usually do – that doesn't necessarily mean that they will do what is necessary to actually accomplish this goal. {...} We need to figure out how to motivate them to create interaction that is good for users."*
>
> Alan Cooper, in *The inmates are running the asylum.*

In this chapter, we investigate the current trends in tools for modeling and designing software applications and interactive systems in particular. We begin describing some successful approaches, categorized according to the class of tool they address. From then on, we compare the most important issues among related tools and, finally, we draw some conclusions as well as some insights for the future generation of interaction design tools.

SECTION 3.1 introduces the theme, stating the importance and amount of effort dedicated to UI development. The surveyed tools are categorized into five different classes, a taxonomy that is employed during this chapter. The clash between the usability and the usefulness of the tools is also discussed in this section.

SECTION 3.2 describes and relates Analysis, Modeling and Design tools, either commercial or open-source.

Model-based UI Design and some of the current tools for supporting it are described in SECTION 3.3. The model-based paradigm is briefly presented, as well as its roots, motivations and the initial tools that appeared in the beginning of the 90's.

SECTION 3.4 is devoted to covering informal tools, i.e. tools based on informal input modalities such as sketching and gestures.

Since interaction design is a team activity, with projects usually involving multiple stakeholders, with multiple interests and distinct backgrounds, we describe collaborative

design tools in SECTION 3.5. We analyze tools that leverage asynchronous and synchronous, located or remote collaboration styles.

SECTION 3.6 covers XML-based languages and tools. XML-based languages have become the standard approach to declaratively specify a UI. UsiXML (Limbourg et al., 2004), AUIML (Azevedo et al., 2000), XIML (Puerta, 2002), and Adobe's MXML and Microsoft's XAML, on the commercial side, are some of the languages analyzed in this section.

SECTION 3.7 describes related work based on a recent CHI Workshop on the Future of UI tools (Olsen and Klemmer, 2005).

The chapter ends with a discussion and conclusions (SECTION 3.8) about the current state of UI-related tools and outlines some directions and requirements tools should implement in order to better adjust to interaction designers and their work.

3.1 Introduction

Nowadays, the role of modeling in the software development process is apparently well agreed-on. Analysis and design models present many advantages to the development process (Constantine, 1994), mainly because they help the programmer organize their ideas and thoughts. They also serve as a succinct and objective language, shared by the group or community of developers.

In a study by (Myers and Rosson, 1992), it has been shown that the amount of effort required to the development of user interfaces represents, in average, 50% of the total effort. Functional requirements specification correspond to 60%, but user interface requirements spends 40% of the total requirements specification effort (Molina, 2003).

In UI tools, there has always been a dichotomy between the sophistication of what can be created (the usefulness of a tool) against the ease of use (the usability). This is related to the *threshold* and *ceiling* of tools (Myers et al., 2000). Threshold is how difficult it is to learn how to use the tool, and the ceiling is how much can be done with it (Myers et al., 2000). Building tools that can provide low threshold and high ceiling at the same time has been referred to as an important challenge (Myers and Rosson, 1992). Figure 3.1 depicts a framework that illustrates the dichotomy between *Usefulness* and *Usability*. These two forces often collide: useful tools tend to suffer from lack of usability, whereas usable tools aren't very useful (only allow the creation of simple artifacts, such as sketches, or semantic-less models).

Abstract specification of interactive systems can be combined with automatic generation techniques in order to reduce development costs (Molina, 2003). However there are still many issues to address regarding model-based interactive systems design, namely maintenance problems, lack of scalability, lack of integration with business logic, inexistence of robust code generators and lack of standards (Molina, 2003).

In this context, model-based user interface development has been the target of much research during the last decades. However, and despite the success obtained by user interface development tools, approaches based on models never reached the industrial maturity

augured in the 80's. The industry is still dominated by tools that, although leveraging the creativity of experienced designers, also foster the building of bad interfaces.

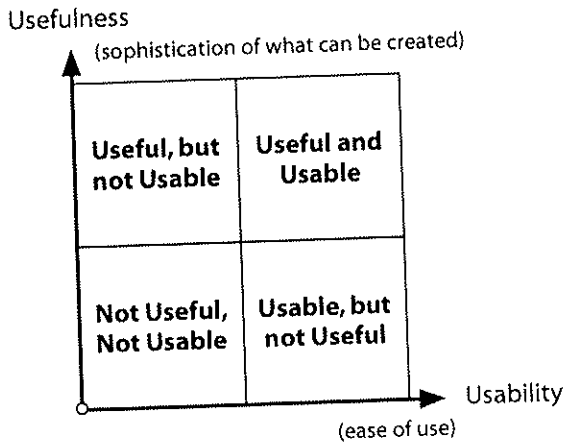


Figure 3.1 Two forces that often collide: useful tools are not usable, usable tools are not sufficiently useful.

The same observation applies to the acceptance of analysis, design and modeling tools on software development. Despite the success and the progress achieved by the emergence of the Unified Modeling Language, modeling tools are still below expectations in terms of impact and industry usage.

Virtually all applications are built using some kind of interface building tool (Myers et al., 2000), and these kind of tools constitute an important segment of the tool market, accounting for more than 100 million $USD per-year (Myers et al., 2000).

What drives the acceptance of a good UI tool and what issues can be addressed in order to build tools that leverage the development of high-quality interactive systems? Before even trying to answer this question, it is imperative to overview the state of the art tools for building and specifying interactive systems. Any tool that supports software development can be generally characterized as a CASE tool (Computer-Aided Software Engineering). However, the software development tradition relates CASE tools with high-threshold, high-ceiling tools that are, to some extent, capable of generating code from high level models. CASE tools can be classified in many perspectives: for instance in terms of

styles of development (RAD - Rapid Application Development), or degree of integration (IDEs - Integrated Development Environments). It is not our purpose here to discuss the classification of CASE tools; instead, we will focus on the following classes of tools, because they more clearly reflect the development tasks involved in any UCD process. We will briefly describe and compare:

- *Analysis, Modeling and Design (AMD) tools.* In the context of this dissertation, these tools include the popular CASE (Computer-Aided Software Engineering) tools and also RAD (Rapid Application Development) tools.
- *Model-based User Interface Design (MB-UID) tools.* These tools involve a systematic approach to UI design usually supported by task models. Tools of this class are particularly suited for interactive design of applications and most of them also focus on automatic generation techniques for the UI creation.
- *Informal tools supporting Sketching and Gesturing.* These are mainly used for the early, creative development stages, when we expect to take advantage from informal input modalities in order to foster exploration of designs, fast communication of ideas and creativity in general.
- *Collaborative Design tools.* Also known as Computer-Supported Cooperative Work (CSCW) tools, these tools are based on recognizing that software design is a highly cooperative task. Therefore, they try to support the communication processes that leverage the cooperation between all stakeholders involved.
- *XML-based Languages and tools.* XML-based markup languages typically work by raising abstraction to the level of a declarative programming model. The tools that use this approach are sometimes regarded as MB-UID tools, but given the growing importance and dissemination of these in industry and academia, we chose to discuss and describe them in a separate section.

3.2 Analysis, Modeling and Design Tools

In this class of tools, we are considering CASE and RAD tools. Popular examples of CASE tools include Rational Rose and UML-based tools in general. RAD tools became widely used with the advent of VisualBasic. Nowadays, they seem to have evolved to other styles, like Adobe's Flash or Dreamweaver. This section is subdivided between the commercial efforts in this area and open-source projects.

3.2.1 *Commercial Tools*

There are many commercially available Analysis, Modeling and Design tools. Most of them are based on the UML. By using UML-based concepts such as requirements, constraints and scenarios to model user interface elements, one expects to build up a sound understanding of user interface behavior without having to plunge immediately into code and other more detailed issues. A typical tool of this kind is MagicDraw (NoMagic, Inc., 2004). Figure 3.2 shows a screenshot of this tool, which is representative of this class of applications.

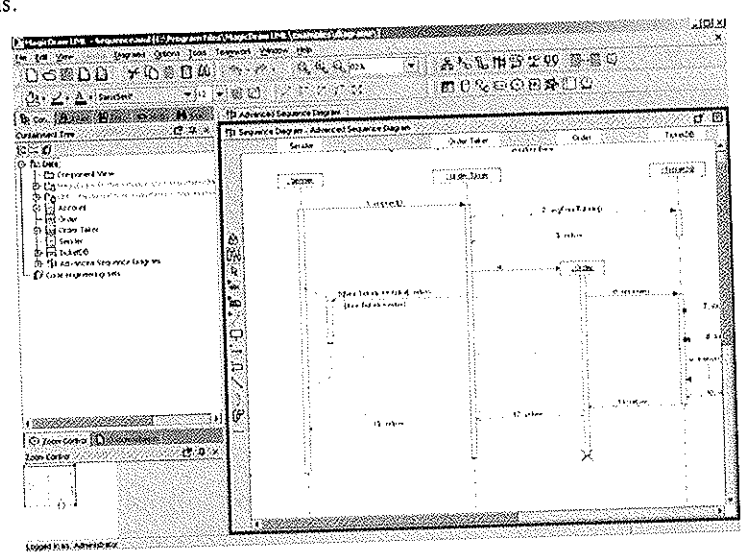


Figure 3.2 Screenshot of MagicDraw showing a sequence diagram.

Rational Rose (IBM, 2004) is probably the most popular modeling tool based on the UML (Rose stands for "Rational Object-oriented Software Engineering"). It is actually a set of tools that support the Rational Unified Process. It provides support for version control, IDE integration, design pattern modeling, test script generation and collaborative modeling environment. An interesting feature of Rose is the ability to publish the UML diagrams as a set of Web pages. This fosters the sharing and distribution of application designs in environments where the Rational Rose tool is not present.

However, Rose lacks support for UI-related activities. The usability of Rose has also been considered very low (Robbins, 1999): the tool is often sold with training. Although Rose follows MS Windows UI Guidelines (Microsoft, 1995), its dialogs are overly complex and diagram editing has to be done through many-tabbed dialogs (see Figure 3.3).

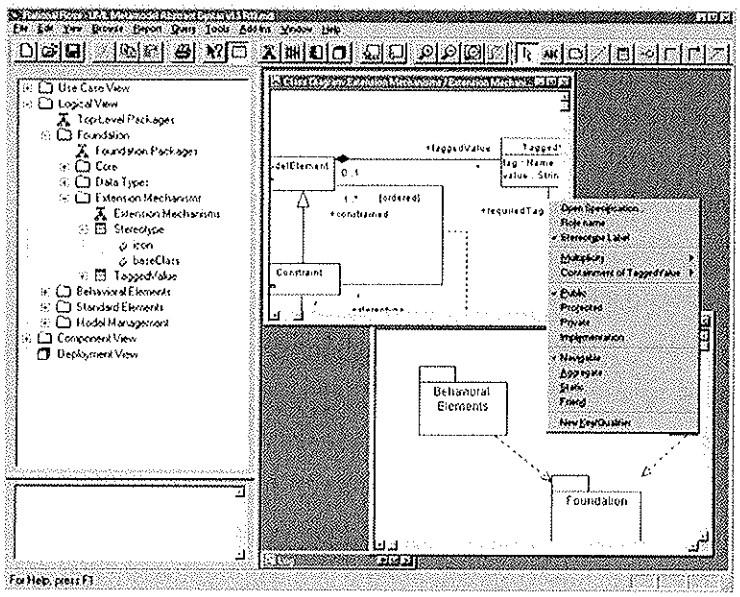


Figure 3.3 The Rational ROSE environment.

Telelogic's (Telelogic, 2004) tools cover all phases of the development process, including requirements handling, analysis, design, implementation and testing.

The Telelogic DOORS/ERS application is used by more than 50,000 users at more than 1,000 companies worldwide. The DOORS/ERS suite is a multiplatform, enterprise tool

designed to capture, link, trace, analyze and manage information throughout the lifecycle of a software development project.

This application allows project managers, software engineers and other stakeholders to work collaboratively on a software project. It enables them to access the same information, often within a distributed environment and on multiple platforms. Telelogic's larger customers, particularly in the Aerospace and Defense industries, use DOORS/ERS to manage large scale software development projects for both UNIX and Windows. It was their customer needs that drove Telelogic's urgent need for cross platform functionality (Telelogic, 2004).

Enterprise Architect is a powerful means by which to specify, document and build software engineering projects (Sparks, 2004). It is based on the UML 2.0 notation and semantics and can also generate and reverse engineer source code in a variety of languages, import database designs from standard data sources and import and export models using the XMI industry standard for tool interoperability.

Ameos from (Aonix, 2004) is also a commercial AMD tool that expects to become "the next generation modeling tool". Recognizing that the wide use of the UML in the market doesn't seem to stop software projects from failing, Ameos proposes a combination of UML 2.0 Profile support, MDA-based Model Transformation and the usage of color in a unique fashion. The Model Management of the UML is an integral part of Ameos and allows distributed working, private workspaces and the configuration of new versions. In order to ensure perfect scalability from small to large projects, Ameos' engineers devised a Multi-User Repository that manages the models created.

Ameos provides UML 2.0 Profile support. Profiles are the standard way to extend the UML and to tailor it to project-specific needs. UML 2.0 describes Profiles and defines how to model them in the UML notation. Ameos offers a Profile Editor that allows stereotypes and tagged values to be defined and assigned to model elements of the UML Metamodel. This ensures that profiles are well designed, documented and easy to use for the entire project team.

Ameos also fosters the use of color to visualize special semantics: the user can assign color to UML Profiles and to Model Elements. Whenever a Model Element is referenced, it

shows up in the assigned color. This usage of color on a semantic level results in UML models which are far easier to read.

In the RAD (Rapid Application Development) category, Visual Studio is definitely the most famous example of tool. This tool is at the lowest level of abstraction regarding UI modeling, since it provides the developer a palette of components that can easily be placed in a screen, thus allowing fast prototyping and testing of the final UI.

However, like Molina (2003) points out, this class of tools is limited, since they only cover the presentation issues (i.e., the dialog issues have to be programmed by the developer). Also, they only address WIMP-based user interfaces (e.g. interactive animations or sprites need also to be programmed by the user). Nevertheless, the most recent release (Visual Basic .NET 2003) already includes integrated support for the creation of applications for wireless, Internet-enabled handheld devices. The user can accomplish these capabilities using his/her previous Visual Basic programming skills.

3.2.2 Open-Source Efforts

The open-source community has not been able to compete with the fast-paced industry of modeling tools. ArgoUML (Robbins et. al, 1997) is one of the very few open-source tools for analysis, modeling and design of software systems. ArgoUML is itself based on an open-source project implementing the UML 1.4 metamodel.

The features included in ArgoUML are all based on Cognitive Theory (Robbins, 1999): knowledge support via critics and checklists, process support via "to-do" lists, visualization support via navigational perspectives. As in all other tools analyzed in this section, ArgoUML does not support any UI-specific modeling.

UI design support for ArgoUML only came with the UMLi project for a UML extension for modeling interactive applications (Silva and Norman, 2003). UMLi conservatively extends UML with explicit support (in the form of an extension to ArgoUML) for interface modeling. Figure 3.4 shows the look of UMLi in the ArgoUML tool.

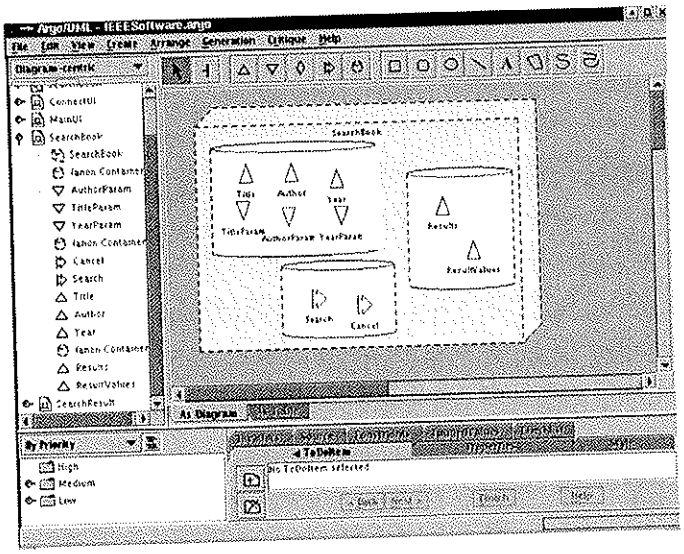


Figure 3.4 The UMLi extension to ArgoUML.

The Eclipse (Eclipse, 2004) project also includes an open-source UML2 module. The whole Eclipse project is aimed at tool integration. This effort is being supported by major tool vendors such as IBM. Eclipse aims to define across-the-board integration standards that will allow different tool vendors to seamlessly work together and provide a cohesive, single development environment.

In terms of UML, however, the effort developed so far has resulted only in a plug-in module that doesn't offer a graphical front-end for easy editing of UML elements.

3.3 Model-Based User Interface Design

Another line of user interface research that has shown significant promise is the issue of automatic techniques for generating interfaces. The goal of this line of work is generally to allow the designer to specify interfaces at a very high level, with the details of the implementation to be provided by the system (Myers, 2000).

Motivations for this include the idea that programmers without user interface design experience could just implement the functionality and rely on these systems to create high quality user interfaces. The systems might allow user interfaces to be created with less effort (since parts would be generated automatically).

Further, there is the promise of significant additional benefits such as automatic portability across multiple types of devices, and automatic generation of documentation for the application.

Automatic and model-based techniques have suffered from the problems of unpredictability. In fact, because heuristics are often involved, the connection between specification and final result can be quite difficult to understand and control. Programmers must also learn a new language for specifying the models, which raises the threshold of use.

The model-based paradigm uses a central database to store a description of all aspects of an interface design. This central description is called the model, and typically contains information about the tasks that users are expected to perform using the application, the data of the application, the commands that users can perform, the presentation and behavior of the interface, and the characteristics of the envisioned users.

HUMANOID (Szekely et al., 1992; Szekely et al., 1995) was one of the very first systems of this kind. It was an early tool for model-based interface design and construction where interfaces were specified by building a declarative description (model) of their presentation and behavior. HUMANOID's modeling language provided simple abstraction, iteration and conditional constructs to model the interface features of these application classes. Its contribution was specifically focused on providing a good declarative language coupled with an easy to use UI. Other early approaches included UIDE (Foley et al., 1991), a very similar system, and ITS (Wiecha et al., 1990). The ITS architecture separated applications into four

layers. The action layer was responsible for implementing back-end application functions. The dialog layer defined the content of the user interface, independent of its style. Content specifies the objects included in each frame of the interface, the flow of control among frames, and what actions are associated with each object. The style rule layer defines the presentation and behavior of a family of interaction techniques. Finally, the style program layer implements primitive toolkit objects that are composed by the rule layer into complete interaction techniques.

3.3.1 *CTT (ConcurTaskTrees Environment)*

ConcurTaskTrees (Paternò, 2000) is a widely accepted notation for specifying Task Models. Fabio Paternò and his team developed a tool to support the notation called CTTe - ConcurTaskTrees Environment (Paternò, 2002, 2005). The notation classifies tasks in four distinct categories:

- User Tasks: these are the tasks undertaken by the user;
- Application Tasks: tasks undertaken by the application, e.g. present a series of search results to the user;
- Interaction Tasks: tasks undertaken by the user through interaction with the system, e.g. pressing a button or choosing a menu item;
- Abstract Tasks: tasks that require the completion of complex activities and that can be decomposed into simpler tasks.

Therefore a task can be decomposed in subtasks. The subtasks are inter-related through the means of temporal operators. The CTT notation is semantically based on the LOTOS (Eijk et al., 1989) specification language. The operators for this language are described in Table 3.1.

Operator	Description
Hierarchy	**Hierarchy.** Tasks at same level represent different options or different tasks at the same abstraction level that have to be performed. Read levels as *"In order to do T1, I need to do T2 and T3"*, or *"In order to do T1, I need to do T2 or T3"*.
T_1 >>T_2	**Enabling.** Specifies second task cannot begin until first task performed. Example: I cannot enroll at university before I have chosen which courses to take.
T_1 [] T_2	**Choice.** Specifies two tasks enabled, then once one has started the other one is no longer enabled. Example: When accessing a web site it is possible either to browse it or to access some detailed information.
T_1 []>> T_2	**Enabling with information passing.** Specifies second task cannot be performed until first task is performed, and that information produced in first task is used as input for the second one. Example: The system generates results only after that the user specifies a query and the results will depend on the query specified.
T_1 ||| T_2	**Concurrent tasks.** Tasks can be performed in any order, or at same time, including the possibility of starting a task before the other one has been completed. Example: In order to check the load of a set of courses, I need to consider what terms they fall in and to consider how much work each course represents.
T_1 |[]| T_2	**Concurrent Communicating Tasks.** Tasks that can exchange information while performed concurrently. Example: An application where the system displays a calendar where it is highlighted the data that is entered in the meantime by the user.
T_1 |=| T_2	**Task independence.** Tasks can be performed in any order, but when one starts then it has to finish before the other one can start. Example: When people install new software they can start by either registering or implementing the installation but if they start one task they have to finish it before moving to the other one.
T_1 [> T_2	**Disabling.** The first task (usually an iterative task) is completely interrupted by the second task. Example: A user can iteratively input data in a form until the form is sent.
T_1 |> T_2	**Suspend-Resume.** The first task can be interrupted by the second one. When the second terminates then the first one can be reactivated from the state reached before. Example: Editing some data and then enabling the possibility of printing them in an environment where when printing is performed then it is not possible to edit.

Table 3.1 The temporal operators in the ConcurTaskTrees notation [taken from (Paternò, 2000)].

Figure 3.5 shows a screenshot of the CTTe tool. We can see a tree composed of tasks concerning an interaction context aimed at searching and selecting objects. An interesting feature of CTTe is the possibility of animating the specification to see if behavior is as expected.

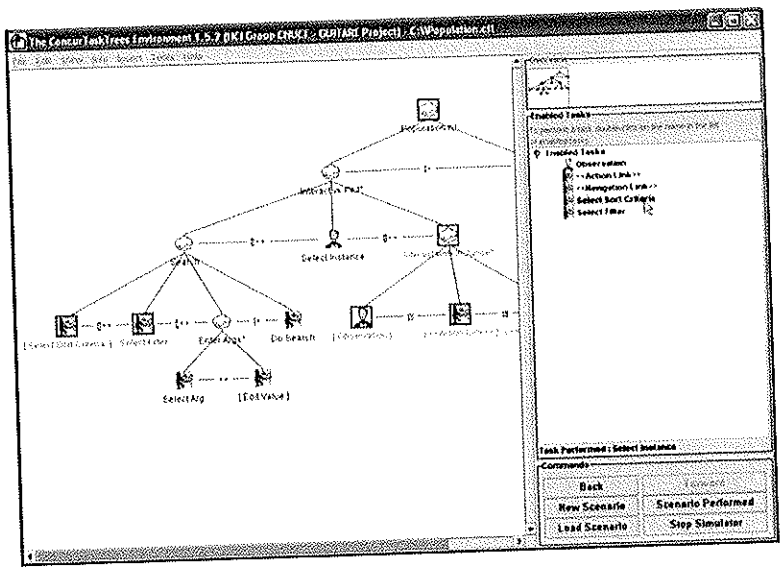


Figure 3.5 CTTe (ConcurTaskTrees environment) screenshot.

3.3.2 *Other CTT-based Tools*

TERESA (Mori et al., 2004) is proposed to present a complete semi-automatic environment supporting a number of transformations useful for designers to build and analyze their design at different abstraction levels, including the task level, and consequently generate the concrete user interface for a specific type of platform. Currently, the tool supports user interface implementations in XHTML and XHTML for mobile devices.

VOICEXML (Bert and Paternò, 2003) and a version for multimodal user interfaces is under development. The tool is able to support different level of automations ranging from completely automatic solutions to highly interactive solutions where designers can tailor or even radically change the solutions proposed by the tool. The last version of the tool supports different entry-points, so designers can start with a high-level task models but they can also start with the abstract user interface level in cases where only a part of the related design process needs to be supported. Using the TERESA tool, the designer can modify the representations while the tool maintains the forward and backward relationships with the other levels thanks to a number of automatic features that have been implemented (e.g. the user can link abstract interaction objects to the corresponding tasks in the task model so that designers can immediately identify their relations). This is useful for designers to

maintain a unique overall picture of the system, with increased consistency among the user interfaces generated for multiple devices.

An evaluation was conducted at the Motorola Italy software development centre (Chesta et al., 2003) with an early version of the TERESA tool. The experiment consisted in developing a prototype version of an e-Agenda application running on both desktop and mobile phone. Results showed similar total times for the traditional and TERESA approaches, with different distributions over the development phases and between time required by the first and the final version. The TERESA-supported method offers a good support to fast prototyping, producing a first version of the interface in a significantly shorter time. However, the time required to modify it increased. The use of the tool almost doubled the required time at the redesign stage, while at development stage the results showed a dramatically improved prototyping performance, reducing the needed time to half. The reported total time increase of using TERESA with respect to using traditional approaches (on average, it was half an hour) is acceptable since it involves a trade-off with design overall quality: many subjects appreciated the benefits of a formal process supporting the choice of the most suitable interaction techniques. For example, designers reported satisfaction about how the tool supported the realization of a coherent page layout and identification of links between pages. The evaluators noticed and appreciated the improved structure of the presentations and more consistent look of the pages resulting from the model-based approach (Chesta et al., 2003).

3.3.3 *MOBI-D (Model-based Interface Designer)*

Puerta and Eisenstein (1999) were among the first to identify the level-of-abstraction mismatch in interface models known as the mapping problem. They argued it was the cause of limitations in the usefulness of model-based approaches (Puerta, 1999). They proposed a general computational framework for solving the mapping problem in model-based systems: the MOBI-D (Model-Based Interface Designer) interface development environment.

MOBI-D allows developers to view and manipulate mapping of abstract task models into concrete interface designs. Exploring the use of this tool, the developer is expected to detect usage patterns that are good candidates for automation.

MOBI-D is actually composed of a set of tools:

- MOBI-D, a design model editor (Domain, Task, Presentation, Dialog and User), conceived for maximum freedom. To set a mapping, a developer simply drags an element of any of the model components and drops it onto the intended target element (see Figure 3.6).
- UTel, a tool that supports requirements elicitation. Based on natural language textual specifications, the tool helps the developer build domain, user and task models, coloring the different localized elements in the text (see Figure 3.7).
- TIMM (The Interface Model Mapper), a decision-support tool that assists developers in navigating the design space of abstract-to-concrete mappings. TIMM can prune the design space of mappings down to a manageable set that the developer can then explore to make final design decisions. It supports mappings from domain-to-presentation, task-to-dialog and task-to-presentation.
- MOBILE (Model-Based Interactive Layout Editor), which is a tool very similar to conventional Interface Builders: it allows editing the composition and layout of interface elements much like a drawing application. However it has major differences: dialog and presentation decisions are guided by the task and user models. The knowledge base given by TIMM constrains design decisions to the most appropriate at each step.

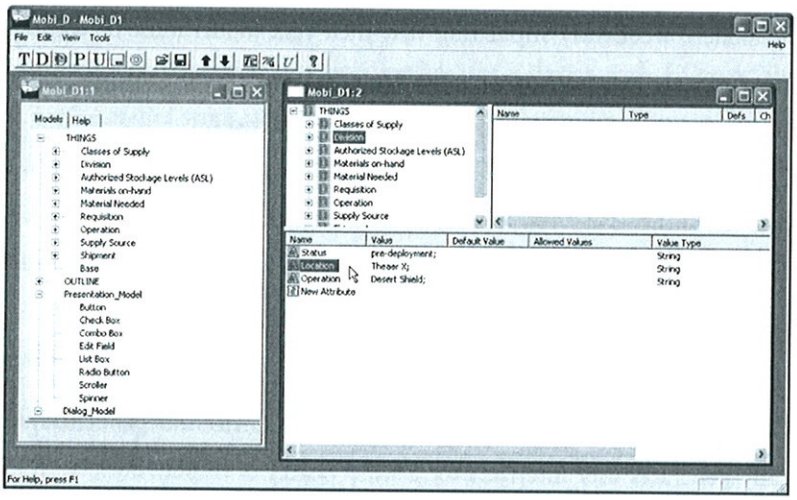


Figure 3.6 MOBI-D screenshot.

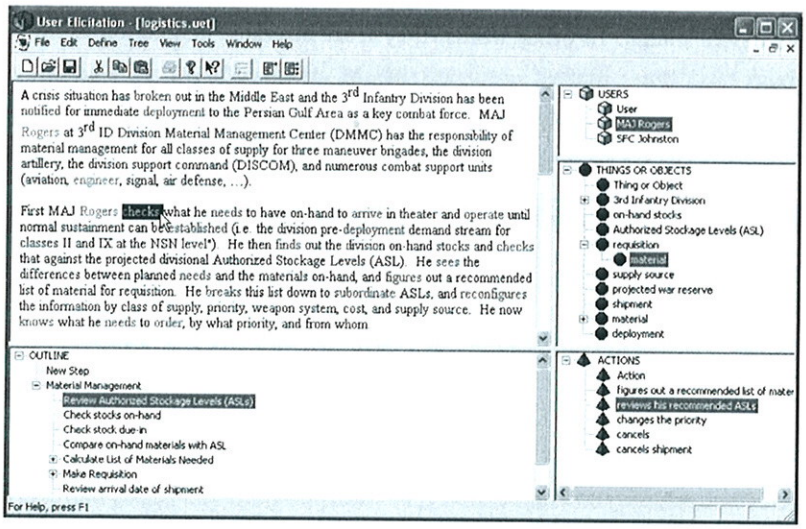


Figure 3.7 UTel screenshot. Notice the colored elements in the text, associated with the models of Users, Domain and Actions.

3.3.4 Teallach

Teallach (Griffiths et al., 1999) is a model-based user interface development environment aimed at database interfaces, an important area where model-based techniques have been rarely applied. MB-IDE's typically support a single fixed method for developing their

component models, frequently stipulating that their task model must be constructed before any other model. This leads to an inflexible interface development lifecycle, imposing a methodology that will not fit into all the developers' practices.

Teallach tries to avoid this difficulty by removing restrictions on the order in which its three models (Griffiths et al., 1999) must be constructed. This flexibility removes restrictions on the order in which links between related model elements can be characterized.

3.3.5 *SUPPLE*

SUPPLE (Gajos and Weld, 2004) is an application and device-independent system, that automatically generates user interfaces for a wide variety of display devices. What makes this approach different from many others is the fact that SUPPLE uses decision-theoretic optimization to render an interface from an abstract functional specification and an interchangeable device model. In other words, they treat interface generation as an optimization problem.

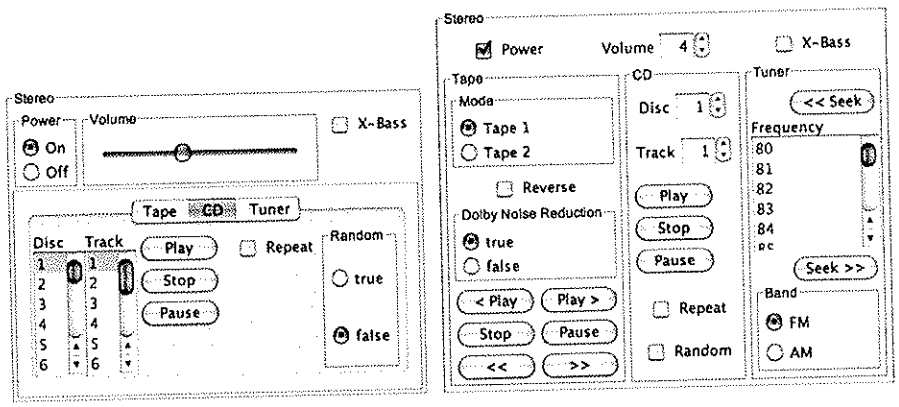


Figure 3.8 An interface to a stereo system rendered on two keyboard and pointer devices of different sizes, using the SUPPLE system [Taken from (Gajos and Weld, 2004)].

SUPPLE also provides mechanisms to adapt and customize the interface elements by changing the appearance, organization and navigational structure of the interface. When asked to render an interface (functionally specified) on a specific device and for a specific user, SUPPLE searches for the rendering that meets the device's constraints and minimizes the estimated cost (user effort) of the person's activity.

Figure 3.8 shows two different interfaces SUPPLE deemed optimal (with respect to a cost function derived from the device and user models) for a device with different screen sizes. Note that unlike earlier work, they not only compute the optimal layout but also choose the individual widgets to be used in the rendering of the UI and the overall navigation structure of the final interface (i.e., placement of parts of the interface in tab panes, pop up windows, etc.).

3.3.6 *Just UI*

The OlivaNova Model Execution System is the one of the first commercially available software systems that generates complete applications from software models. Unlike other software solutions pursuing the promise of Model Driven Architecture (MDA), this "Model Execution System isn't limited to building embedded systems (that lack GUIs), database infrastructure, or integration plumbing" (Care Technologies, 2006). Instead, the approach followed by the OlivaNova Model Execution System takes class models, functional models, and presentation models and creates a completely functional and executable software application.

Just UI is a pattern-based development method, with appropriate tool support, where the designer starts by specifying a CTT model (see §3.3.1) and a domain model, expressed as a UML class diagram, using the OlivaNova Model Editor (Molina et al., 2002). A pattern-based approach is applied based on the semantic information contained in these models.

The specification obtained with the Just UI approach is platform independent. The specifications don't hold any design details. This means that the specification can be reused to provide similar UIs across several target platforms. At the same time, the concepts used have direct translation to the final implementation. Each presentation pattern is reified (mapped) to windows, forms, web pages or any other implementation of presentation units that can be presented in the solution domain (Molina et al., 2002).

3.4 Sketching and Gesturing

Electronic sketching traces its roots to Sutherland's original Sketchpad (Sutherland, 1963) which pioneered the use of a stylus (in Sketchpad's case an electronic light-pen) to draw on one of the first graphical displays. Stylus-based graphical drawing and the potential benefits of pen-based computer interfaces have been studied in various research efforts since then (Negroponte and Taggart, 1971; Negroponte, 1973; Wolf et al., 1989; Brocklehurst, 1991) and also in some pioneering commercial efforts (Wang, 1988; GO 1992; Microsoft, 1992; Apple, 1993). For a comprehensive survey of the history of pen interfaces in research, see (Landay 1996; Long 2001).

Prototyping interfaces with electronic sketching tools has also proven successful in systems such as SILK (Landay and Myers, 2001) or DENIM (Newman et al., 2003). Sketching is believed to be important during the early stages of prototyping, because it helps the designers' creative process: the ambiguity of sketches with uncertain types or sizes encourages the exploration of new designs without getting lost in the details, thus forcing designers to focus on important issues at this stage, such as the overall structure and flow of the interaction (Landay and Myers, 2001).

In the following subsections, we will review some of the current UI tools that are based on sketch input.

3.4.1 *DEMAIS*

DEMAIS (Bailey et al., 2001; Bailey, 2002) is an informal tool for multimedia authoring. It includes the concept of joined formal and informal representations, in which audio and video clips co-exist with sketched representations, claiming that informal representations have specific benefits (Bailey and Konstan, 2003). It also includes the concept of rich transitions between scenes in its storyboard based on mouse events. As a multimedia tool, DEMAIS is focused on the design of rich, diverse output.

3.4.2 *DENIM*

Newman and Landay (2000), through a study already partly described in this thesis (see Chapter 2 , §2.3.2), discovered which tools were used to produce which artifacts by the designers who participated in their study, which we reproduce here in Table 3.2. This table also shows the phases in which the artifacts were most prevalent and which design focus each artifact is most related to. Designers were observed to sketch while producing site maps, storyboards, schematics, and mock-ups. All of these artifacts were later reproduced in a more formal state, except for storyboards.

Each design artifact is more relevant in certain phases than in others, and some strongly focus on certain aspects of the site design rather than others. Table 3.2 lays out the major classes of artifacts observed during the study, the phases in which they were most relevant, the aspect of design on which they focused, and the tools most commonly used to create them (*italics* indicate the most commonly used tools for each artifact).

Artifact	**Phase**	**Focus**	**Tools**
Site map	Discovery, Exploration	Information, Navigation	*Sketching*, *Visio*, Illustrator
Storyboard	Exploration	Navigation	*Sketching*, Illustrator
Schematic	Exploration, Refinement	Information, Navigation	*Sketching*, *Illustrator*, Visio
Mock-up	Refinement	Graphic	*Photoshop*, HTML, Sketch
Prototype	Refinement, Production	All	*HTML*, Director
Specification	Production	All	*Word*, HTML

Table 3.2 Artifacts used by web site designers along with phase and tools used.

Based on their study of web site design practices, Newman and colleagues proposed a sketch-based tool called DENIM supporting information and navigation design of web sites (Newman and Landay, 2000; Newman et al., 2003).

DENIM (see Figure 3.9), supports sketching input, allows design at different refinement levels, and unifies the levels through zooming. DENIM supports visualizations matching the site map, storyboard and schematic (see Table 3.2). Another interesting feature of DENIM is that it allows users to interact with their site designs through a "run mode" (Newman et al., 2003) which displays the sketched pages in a limited functionality "browser" that allows the user to navigate the site by clicking active regions of the sketches and linking to other pages within the site.

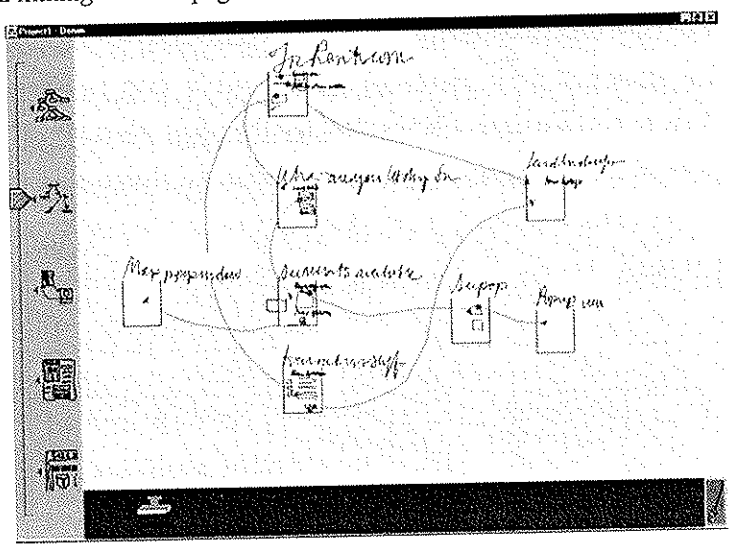


Figure 3.9 The DENIM tool [taken from http://dub.washington.edu/denim/].

However, widget recognition is hard for these systems (Landay and Myers, 2001), since any widget recognition algorithm might be too error-prone. Also, usability tests reported that some users had trouble manipulating and entering text, and understanding how to select, group and move objects.

3.4.3 *The Designer's Outpost*

In a common early-phase practice, designers collect ideas about what should be in a web site onto Post-it notes and arrange them on the wall into categories. This technique, often called affinity diagramming (Beyer and Holtzblatt, 1998), is a form of collaborative sketching used to determine the site structure. Klemmer and colleagues developed a tool,

The Designers' Outpost, that tries to combine the advantages of both paper and electronic media (Klemmer et al., 2001).

In Outpost, users collaboratively author web site information architectures on an electronic whiteboard using physical media (Post-it notes and images), structuring and annotating that information with electronic pens, as shown in Figure 3.10.

Figure 3.10 Using the Designer's Outpost [taken from http://guir.berkeley.edu/projects/outpost/].

This interaction is accomplished through a sophisticated hardware architecture which is partially illustrated in Figure 3.11. A touch-sensitive SmartBoard is augmented with a robust computer vision system, employing a rear-mounted video camera for capturing movement and a front-mounted high-resolution camera for capturing ink.

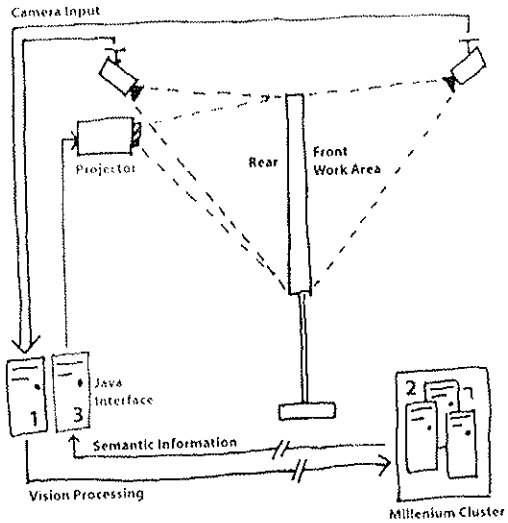


Figure 3.11 The (quite complex) hardware architecture which is behind the Designers' Outpost [taken from http://guir.berkeley.edu/projects/outpost/].

3.4.4 *Ideogramic UML (Knight tool)*

Damm et al. (2000) describe a tool, called Knight, that is based on a direct, whiteboard-like interaction achieved using gesture input on a large electronic whiteboard. Damm et al. (2000) argue that the conflicting advantages and disadvantages of whiteboards and modeling tools can lead to frustrating and time consuming switches between the two technologies. Therefore, they aim at offering a tool capable of offering the best of both worlds.

The Knight tool (see Figure 3.12) uses an electronic board that naturally supports collaborative work, since several persons can work around it. Knight uses gestures as the main input mechanism. These are used for text input as well as for creating and modifying other diagram elements. By sketching boxes and lines on the screen, elements of UML class diagrams can quickly be created and manipulated, through an interaction similar to that of a traditional whiteboard.

The idea of using gestures in modeling is also useful with other input devices, such as graphic tablets and PDA's.

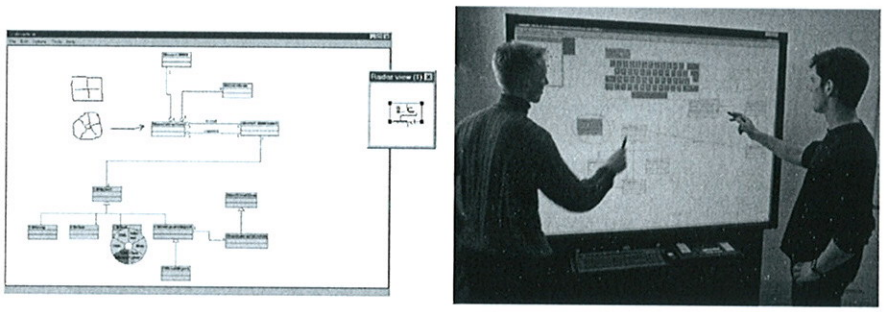


Figure 3.12 The Knight user interface (left) and using Knight on a whiteboard (right). [Taken from (Damm et al., 2000)].

The Knight research project originated a commercial product called IdeogramicUML (Hansen and Ratzer, 2002). They discuss how this tool was effectively used to teach object-oriented modeling.

3.4.5 SketchiXML

Recognizing the importance of supporting sketching activities during the early phases of design, Coyette and Vanderdonckt (2005) presented a tool called SKETCHIXML that enables designers to sketch user interfaces in different levels of detail and supporting different contexts of use. This project identified a series of shortcomings about current sketching tools, such as:

- lack of support for reusability of the output produced;
- binding to a particular programming language (which also prevents reuse of designs if the designer switches the target platform);
- lack of flexibility because the designer doesn't get to choose the exact timing of the sketch recognition;
- always imposing the same sketching scheme.

SKETCHIXML is therefore presented as "a new informal prototyping tool solving all these shortcomings, letting designers sketch user interfaces as easily as on paper." (Coyette and Vanderdonckt, 2005). The tool also provides the designer with on-demand critiquing and assistance, and it generates code in UsiXML (User Interface eXtensible Markup Language, described in § 3.6.1), a platform-independent User Interface Description Language

(UIDL), which can be exploited to construct code for one or several UIs, and for one or many contexts of use simultaneously.

Another interesting aspect of this research is related to a study aimed at surveying the users' personal preferences for choosing a "sketch widgets" library, amongst the possibility of different sketches. The survey was applied to 60 users from different activity sectors with different backgrounds, in order to identify how these people would intuitively represent the widgets to be handled by SKETCHIXML. The built-in collection of sketches in SKETCHIXML was chosen according to the results of this study. Additionally, the set of representations is not hard coded and can be reconfigured by the user through an external configuration file, thus increasing the flexibility of the tool.

3.4.6 *Java SketchIt*

In a similar effort aimed at increasing the prototyping speed, the tool Java SketchIt (Caetano et al., 2002) allows designers to quickly sketch widgets which are automatically recognized by the tool. Java Swing UI code is automatically generated. The resulting UI layout can be beautified using an a posteriori set of grammar rules. Usability studies demonstrated that users found the system to be more comfortable, natural and intuitive to use than a commercial visual interface builder (Caetano et al., 2002).

A successor tool, Java SketchIt 2, is being developed in order to overcome some of its predecessor disadvantages, particularly the limited combination of two gestures for each of the recognized widgets.

3.5 Collaborative Design

As we have already mentioned, software design (which includes Interaction Design) is often a team activity and most projects involve stakeholders with different backgrounds that must cooperate in many different and interrelated activities.

Although any collaborative design activity involves communication and coordination, software design has an extra complicating factor: the product being designed is an incomplete description, not a tangible concrete object. Software designers can envision user interfaces by writing prototypes and storyboards. But the end product is inevitably more abstract and difficult to describe or portray (Potts and Catledge, 1996).

This has motivated the development of tools that support cooperation in software design (Wu, 2003). There are a wide variety of tools that are used to support collaboration in software design. From traditional, office communication tools such as a telephone or e-mail, electronic collaboration tools (Bly and Minneman, 1990; Roseman and Greenberg, 1996) supporting synchronous communication and artifact sharing, as well as specific software design tools that provide traditional CASE functionality along with some degree of integrated support for collaboration (Graham et al., 1999). Other tools support informal media, like the ones described in the previous section, to facilitate brain-storming and early design-thinking as well as synchronous collaboration. Other tools focus on asynchronous artifact sharing (Cocreate, 2004) as a mechanism to support collaboration.

The study by Wu et al. (2003) showed that designers frequently change their physical location throughout the day, in order to support enhanced communication. Designers also change the ways in which they communicate, changing modalities and styles. Wherever co-located interaction was possible, there was a great preference for and use of face-to-face communication, often in conjunction with whiteboards.

This study motivated a discussion about the resulting impact on tool design. In particular, it noted that a tool supporting only asynchronous communication, via e-mail or document repositories does not address the predominantly synchronous interactions in which designers engage (Wu et al., 2003).

Recognizing that designers work together in a variety of styles and move frequently between these styles throughout the course of their work, Wu and Graham propose a tool called the Software Design Board (SDB), a collaborative design tool that supports a variety of styles of collaboration and facilitates transitions between them (Wu and Graham, 2004).

The Software Design Board was developed in order to support the following aspects:

- freehand creation of syntactically correct UML diagrams by using stylus-based input or an enhanced whiteboard (also known as smartboards);
- integration of existing applications into all workstyles (any design document from any application may be embedded into some area of the board);

The tool also supports transitions between asynchronous and synchronous styles of collaboration, and between co-located and distributed styles of collaboration. The whiteboard space can be divided into any number of segments. These segments allow data to be shared in different ways. A segment is an area in the board containing contextually related data. As with a regular whiteboard, a user explicitly specifies the segmentation of data in the board through delineating strokes, e.g. a surrounding box or circle. Segments can be shared with others to allow users of other SDB clients to connect and synchronously interact with each other and share data. To share segments asynchronously, another client connects and copies the content of the segment to his/her local client. This data can then be manipulated without affecting the original segment. If possible, diverging copies of segments may be manually or automatically reconciled.

Gesture information is automatically transmitted between synchronously shared segments via telepointers. In Figure 3.13 we can see two users (Nick and Baha) sharing a segment in SDB (in the left). Nick is also sharing a different segment with James (right).

Few commercial tools support synchronous, distributed collaboration to some extent. One of them is Team Konesa, previously known as Citerra (CanyonBlue, 2004). This is a collaborative version of Konesa Modeler. Using UML and a synchronous collaboration, the development teams maintain vision-focused interaction regardless of tangible boundaries. Members are able to work on individual tasks, but artifacts are stored within the team model.

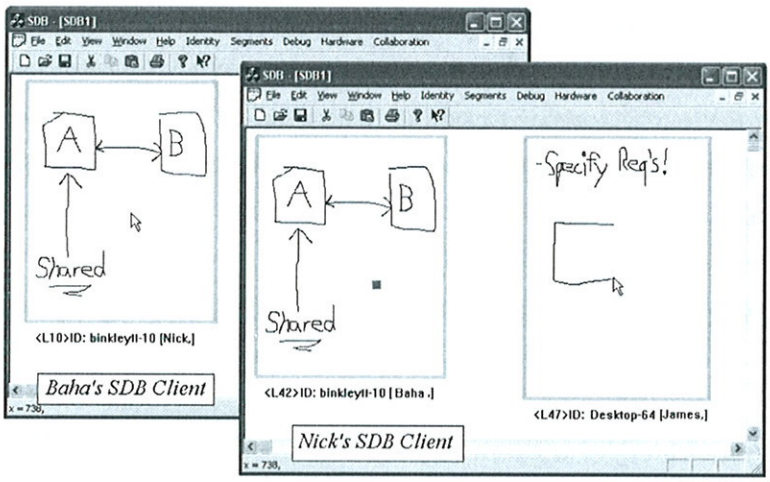


Figure 3.13 Two users (Nick and Baha) sharing a segment in Software Design Board (in the left). Nick is also sharing a different segment with James (right). Taken from (Wu and Graham, 2004).

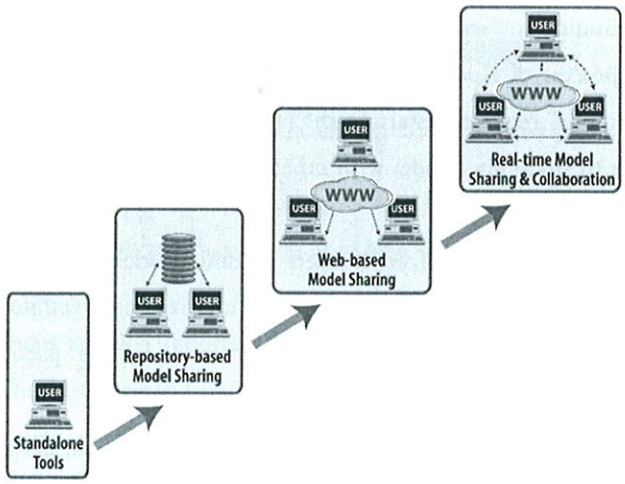


Figure 3.14 Evolution of UML modeling tools [Taken from (Canyonblue, 2004)].

This collaborative model facilitates knowledge transfer and skills exchange across the development team, enabling developers, who may be new to UML development, to easily tap into the team's collective expertise. Canyonblue identifies a market segment for tools capable of combining the benefits of UML modeling with true real-time collaboration,

considering that class of tools as the natural next step in the evolution of UML modeling tools (see Figure 3.14).

In order to create a cooperative model in Konesa, one of the developers must create a model and prepare it by: (a) naming it so that others can locate it; (b) switching collaboration on; (c) assigning write privileges to the other participants. In case of early creative processes, where people frequently brainstorm, this process is too arduous.

To support collaboration, this tool provides an activity log, that shows the actions performed on the model, and a bulletin board for traditional instant messaging between the developers. It provides very little awareness about what other participants are doing. The only "awareness" provided is shown by locking elements when another participant has opened the property dialog for a given model element. Thus, it is more of a concurrency control mechanism rather than an awareness support feature.

Another tool, Embarcadero's Describe (Embarcadero, 2004) has two modes: "local" and "server". When running in "server mode", all models are shared on a central server, and the initial setup process of Konesa is not necessary. However, since shifting from "local mode" to "server mode" requires restarting the program, the user has to decide in advance, whether he wants to share his model with others or not, meaning that ad-hoc collaboration is cumbersome.

There is even less awareness of other users in Embarcadero Describe than in Team Konesa. There is no activity log and bulletin board, and it is not even possible to see who else is working on the model. Changes to an element made by one participant are visible to the others, as soon as the element is deselected.

The Knight tool (Damm, 2000) described in the previous section, has a distributed version that was developed having all these issues in mind. For instance, collaboration in Distributed Knight is based on the notion of "sessions". Two or more instances of Distributed Knight engaged in the same session will contain the same data, and changes performed in one instance will be immediately visible in the others. Once a session has been created, others can join it (Damm, 2000).

Awareness mechanisms are implemented in Knight using visual cues in the drawing surface (a technique known as workspace awareness). Since there may be more than two clients in the same session, each client has its own color that is used for all visual cues. For

instance, to see where other participants are looking at (in the diagram), this tool uses a radar view for showing the current user's viewport as well as other participant's viewports.

Also, the mouse pointer can be used for pointing or calling other participant's attention. Distributed Knight supports this by displaying the mouse cursors of other users. And since the primary interaction means in Distributed Knight is drawing gestures in the drawing surface, they also decided to show the other participants' gestures. For instance, Figure 3.15, depicts a remote user drawing a gesture for creating a UML Class (in light gray).

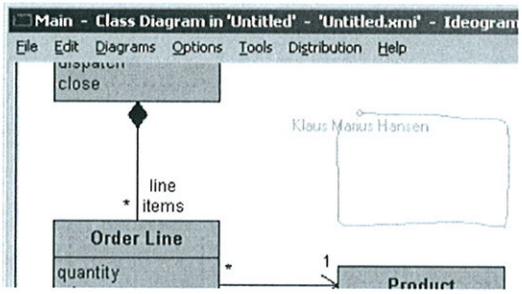


Figure 3.15 A remote user drawing a gesture for creating a class in Distributed Knight.

In a usability test, Damn and colleagues report that few users had trouble with UML or the tool itself and some reported that it was engaging to participate in a distributed modeling session. All participants found the workspace awareness properties to be sufficient, and, what is perhaps equally important, it was not considered disturbing for their work (Damn, 2000). The participants reported that they were able to monitor the collaborator's work and to coordinate their own work. Observations showed that the participants were rather reluctant to modifying each others' elements, i.e., there was a sense of ownership of the individual elements.

The authors note that although the Knight tools are aimed at object-oriented modeling, the issues raised and lessons learned are also applicable to user interface design, or even business process reengineering.

3.5.1 The Distributed Designers' Outpost

Following the previous work on the *Designers' Outpost* installation for UI design (Klemmer et al, 2001), Everitt et al. (2003) developed the *Distributed Designers' Outpost*, a remote collaboration system based on the *Designers' Outpost*.

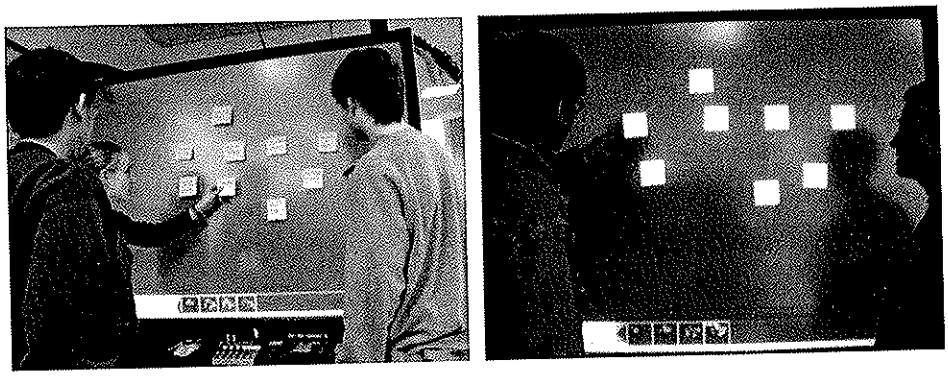


Figure 3.16 The Distributed Designers' Outpost: a very interesting system, because post-it notes are physical (on the left) and electronic (on the right). The bottom of the SmartBoard screen shows Outpost's history bar showing previous states of the system.

The system was informally evaluated with six professional designers. Designers were "excited by the prospect of physical remote collaboration but found some coordination challenges in the interaction with shared artifacts" (Klemmer et al, 2001).

Everitt et al. (2003) present and evaluate two mechanisms for awareness: transient ink input for gestures and a blue shadow of the remote collaborator for presence. The transient ink is a pen-based interaction technique for conveying deictic (pointing) gestures. Users mark up the board to suggest changes or relationships without permanently cluttering the workspace. Transient ink is displayed on both boards for a few seconds, then fades away.

The mechanism for presence awareness is a blue shadow that represents the location of the remote participants with respect to the shared workspace. Users of the system can get a sense of the locations and intentions of remote collaborators without needing their physical presence.

This work was based on previous field studies, where four ways of remote collaboration between designers were found: Whiteboard, video, and e-mail; Two whiteboards and videoconference; Collocated meetings (and occasional conference call); and using Visio and e-mail.

3.5.2 CooLDev: a Collaboration tool based on Activity Theory

Inspired by Activity Theory (described in CHAPTER 2, §2.2.1 in this thesis), Lewandowski and Bourguin (2005) have recently tried to accomplish new mechanisms to support collaboration in software development using the Eclipse platform (Eclipse, 2004).

Eclipse presents the user with perspectives and a perspective corresponds to a particular point of view on the working environment (and the activated plug-ins) during the achievement of a task. It manages the plug-ins activation and arrangement at the user interface level. Eclipse lets the user create and modify his own perspectives, thus saving his preferences for a task. From Lewandowski and Bourguin's (2005) point of view, the perspectives mechanism "provides a powerful mean to crystallize some experience. However, this experience is not intended to be shared by users. Even if some people may work with the same perspective because it has been packaged with a specific plug-in, nothing is provided for sharing perspectives in the context of a particular global, evolving and cooperative activity" (Lewandowski and Bourguin, 2005).

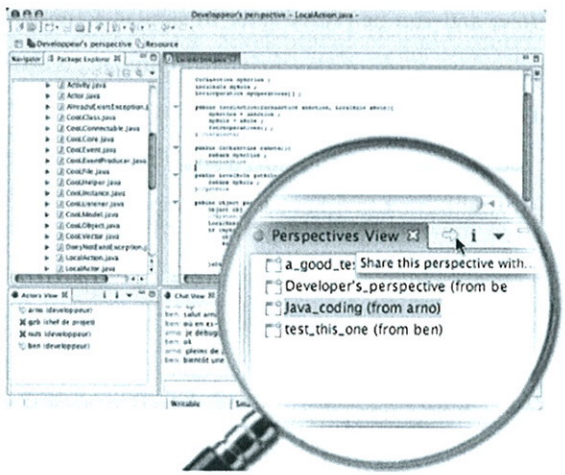


Figure 3.17 CooLDev environment, and zoom on the shared perspectives view.

As a first step in trying to better manage what they call *experience crystallization* they developed the CooLDev tool as an Eclipse plug-in, as well as some basic features over the perspective mechanisms.

CooLDev associates roles in a given activity with particular perspectives. When a subject joins an activity, he retrieves a perspective instance that is defined according to its role. However, all the subjects playing the same role may not share exactly the same perspective since they can adapt/modify it according to their role and emerging needs. These instances, originated from the same role model, can then evolve and be considered as prototypes reflecting the subject's experience he has developed while playing his role. There is also a plug-in, presented in Figure 3.17, allowing subjects to share their perspectives. The view shows the shared perspectives, and allows the users to test these shared perspectives. CooLDev also allows generalizing a perspective at the task level, i.e. in a role model, after negotiations between the subjects. Following the co-evolution principle, this form of co-construction helps the subjects to develop a real experience that is written into the perspective prototype. This experience can be crystallized in the model that can benefit to the subjects playing this role, and can be reused later in similar activities. This demonstrates how a community of developers can make their environment co-evolve by sharing their experience. As CooLDev also supports transformations of the whole activity (process) model, Lewandowski and Bourguin are currently extending this prototype-based approach to support experience crystallization in other activity support elements.

3.6 XML-based Languages and Tools

XML-based UI markup languages typically work by raising abstraction to the level of a declarative programming model. In this section, we will briefly cover the most significant languages (and sometimes the corresponding supporting tools) that are based on XML.

3.6.1 UsiXML: Language and Tools

UsiXML is a XML-compliant markup language that describes the UI for multiple contexts of use (Limbourg et al., 2004).

UsiXML consists of a User Interface Description Language (UIDL). A UIDL captures the essence of what a UI is or should be independently of physical characteristics.

UsiXML describes at a high level of abstraction the constituting elements of the UI of an application: widgets, controls, containers, modalities, interaction techniques. It was designed specifically to support what has become known as multi-path development (Limbourg et al., 2004). UsiXML supports the Cameleon Reference Framework a process that defines a series of steps towards the development of multi-context interactive systems. Figure 3.18 illustrates these steps for two hypothetical contexts of use (A and B). These steps are the following:

1. Final UI (FUI): is the operational UI i.e. any UI running on a particular computing platform either by interpretation (e.g., through a Web browser) or by execution (e.g., after compilation of code in an interactive development environment).

2. Concrete UI (CUI): concretizes an abstract UI for a given context of use into Concrete Interaction Objects (CIOs) (Vanderdonckt and Berquin, 1999) so as to define widgets layout and interface navigation. It abstracts a FUI into a UI definition that is independent of any computing platform. Although a CUI makes explicit the final Look & Feel of a FUI, it is still a mock-up that runs only within a particular environment. A CUI can also be considered as a reification of an AUI (Abstract User Interface, which is described in the next item) at the upper level and an abstraction of the FUI with respect to the platform.

3. Abstract UI (AUI): defines interaction spaces (or presentation units) by grouping subtasks according to various criteria (e.g., task model structural patterns, cognitive load analysis, semantic relationships identification), a navigation scheme between the interaction spaces and selects Abstract Interaction Objects (AIOs) (Vanderdonckt and Berquin, 1999) for each concept so that they are independent of any modality. An AUI abstracts a CUI into a UI definition that is independent of any modality of interaction (e.g., graphical interaction, vocal interaction, speech synthesis and recognition, video-based interaction, virtual, augmented or mixed reality). An AUI can also be considered as a canonical expression of the rendering of the domain concepts and tasks in a way that is independent from any modality of interaction. For example, in ARTStudio (Calvary et al., 2001) an AUI is a collection of related workspaces. The relations between the workspaces are inferred from the task relationships expressed at the upper level (task and concepts). An AUI is considered as an abstraction of a CUI with respect to modality.

4. Task & Concepts (T&C): describe the various tasks to be carried out and the domain-oriented concepts as they are required by these tasks to be performed. These objects are considered as instances of classes representing the concepts manipulated.

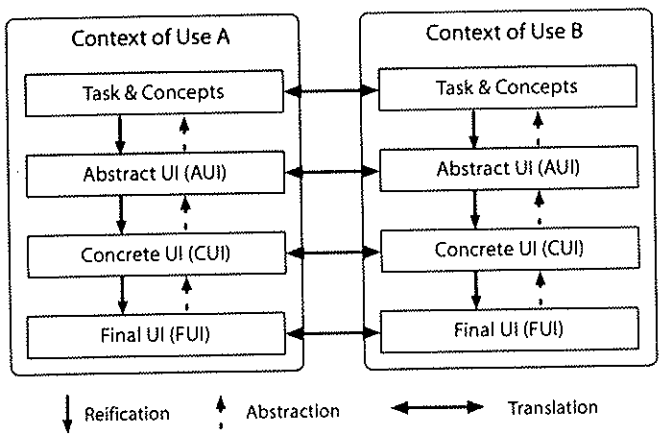


Figure 3.18 Illustration of the Cameleon Reference Framework (Limbourg et al., 2004).

This framework exhibits three types of basic transformation types: (1,2) Abstraction (respectively, Reification) is a process of elicitation of artifacts that are more abstract (respectively, concrete) than the artifacts that serve as input to this process. Abstraction is the

opposite of reification. (3) Translation is a process that elicits artifacts intended for a particular context of use from artifacts of a similar development step but aimed at a different context of use. With respect to this framework, multi-path UI development refers to a UI engineering method and tool that enables a designer to (1) start a development activity from any entry point of the reference framework (Figure 3.18), (2) get substantial support in the performance of all basic transformation types and their combinations of Figure 3.18.

Beyond the AUI and CUI models that reflect the AUI and CUI levels, the other UI models are defined as follows:

- uiModel: this is the topmost superclass containing common features shared by all component models of a UI. A uiModel may consist of a list of component model in any order and number, such as task model, a domain model, an abstract UI model, a concrete UI model, mapping model, and context model, which are inherited from uiModel.
- taskModel: this is a model describing the interactive task as viewed by the end user interacting with the system. A task model represents a decomposition of tasks into subtasks linked with task relationships. Therefore, the decomposition relationship is the privileged relationship to express this hierarchy, while temporal relationships express the temporal constraints between sub-tasks of a same parent task. A task model is expressed according to the ConcurTaskTree notation (Paternò, 2000).
- domainModel: is a description of the classes of objects manipulated by a user while interacting with a system.
- mappingModel: is a model containing a series of related mappings (i.e., a declaration of an inter-model relationship) between models or elements of models. A mapping model serves to gather a set of inter-model relationships that are semantically related.
- contextModel: is a model describing the three aspects of a context of use in which a end user is carrying out an interactive task with a specific computing platform in a given surrounding environment. Consequently, a context model consists of a user model, a platform model, and an environment model.

There are a myriad of tools currently supporting UsiXML. Most of them can be freely downloaded from www.usixml.org. Here we will only describe a few (in §3.4.5 we already described a sketching tool based on UsiXML).

IDEALXML. IDEALXML (Montero et al., 2005) is a model-based user interface development environment using UsiXML. This tool offers two facilities: on the one hand, it is an editor / manager of patterns of any type (interaction, collaboration, design, etc.) which are stored using the PLML format; and on the other hand, it is a development environment that assists software engineers with pattern-based experience in which usability is considered.

REVERSIXML. The REVERSIXML tool (Bouillon et al., 2004), automatically reverse engineers the presentation model of an existing HTML Web page at both the CUI and AUI levels, with or without intra-model, inter-model mappings. This tool allows developers to recover an existing UI so as to incorporate it again in the development process. In this case, a re-engineering can be obtained by combining two abstractions, one translation, and two reifications.

GRAFIXML. What distinguishes GRAFIXML from other UI graphical editors are its capabilities to directly generate UsiXML specifications at the different levels of abstractions represented in Figure 3.18: FUI, CUI (with or without relationships), and AUI (with or without relationships). In addition, a UI can be saved simultaneously with CUI and AUI specifications, while establishing and maintaining the inter-model relationships.

3.6.2 AUIML

AUIML (Abstract User Interface Markup Language) (Azevedo et al, 2000) is another declarative language, based on HTML, where the main concern was to separate presentation elements from the interaction semantics. AUIML was used along with the Ovid and Wisdom methodologies to allow UI specification at an abstract level.

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 automatically translating an AUIML document into a concrete user interface).

3.6.3 XUL

XUL (eXtensible UI Language) (Ginda, 2000) is actually a multi-platform UI technology based on Mozilla's XML (Mozilla, 2006). It allows designers and developers to specify a multi-platform GUI using a mixture of XML, HTML, CSS and JavaScript. One can modify any aspect of the UI from a Mozilla application (based on XUL) simply by modifying files that use the standard syntax of a web page.

3.6.4 XIML and UI-Pilot

XIML (eXtensible Interface Markup Language) (Puerta, 2002) is another language for supporting multiple models of UI specifications. An XIML specification can lead to a run-time interpretation or to a design-time code generation phase. Using a model-based development process, associated to XIML, the user defines a model of the UI through five models describing tasks, domain, users, dialog and presentation, together with all inter-model associations and dependencies.

Supporting this language, the User Interface Pilot (Puerta et al., 2005) is a software tool that specifically tries to guide the early design of a user interface. It adopts an engineering perspective in UI design, and enables designers to create what Puerta calls abstract wire-frames. Each wire-frame represents a page in a website or a screen on a desktop or mobile application. They are non-functional prototypical depictions of the layout and widgets of a page or screen.

UI Pilot includes three palettes with tasks, data items, and user type objects that designers first populate and then designate as elements of a wire-frame. UI Pilot has been used in over twenty real-world user interface design projects (Puerta et al., 2005).

3.6.5 Microsoft XAML

Avalon, part of the next version of Windows, code-named "*Longhorn*", consists mostly of a new collection of classes added to the .NET Framework. Avalon also defines a new markup language you can use in Longhorn that's code-named XAML - Extensible Application Markup Language (pronounced "zammel"). XAML is used much like HTML to define a layout of text, images, and controls. Being based on XML, XAML has stricter and much

less ambiguous syntax than HTML. With this lightweight markup language for UI's, Microsoft's strategy is also addressing the need for convergence and unification of both Web and Desktop programming models.

Most applications written to Avalon will probably contain both program code and XAML. XAML can be used for defining the initial visual interface of an application, and the developer then writes code for doing everything else. One can embed the program code directly in XAML or keep it in a separate file. Everything that is possible to do in XAML can also be done in program code, so it's possible to write a program without using any XAML at all. The reverse is not true, however; there are many tasks that can only be done in program code, so only the simplest applications will consist entirely of XAML.

Figure 3.19 Hello World XAML sample code, as a Web page (top) and Windows Program (bottom).

Figure 3.19 shows an example of a XAML program, rendered for both Web and Windows Desktop. The rendering is obtained from a simple XAML code which declares a simple "Hello World":

```
<TextPanel xmlns="http://schemas.microsoft.com/2003/xaml"
        Background="BlanchedAlmond"
        FontFamily="Comic sans MS"
        FontSize="36pt"
        HorizontalAlignment="Center">
    Hello, world!
</TextPanel>
```

The equivalent C# code is the following:

```
using System;
using MSAvalon.Windows;
using MSAvalon.Windows.Controls;
using MSAvalon.Windows.Media;

class HelloWorldApp: Application
{
    [STAThread]
    static void Main()
    {
        HelloWorldApp app = new HelloWorldApp();
        app.Run();
    }
    protected override void OnStartingUp(StartingUpCancelEventArgs args)
    {
        Window win = new Window();
        TextPanel tp = new TextPanel();

        tp.Background = Brushes.BlanchedAlmond;
        tp.FontFamily = "Comic sans MS";
        tp.FontSize = new FontSize(72f, FontSizeType.Point);
        tp.HorizontalAlignment =
                        MSAvalon.Windows.HorizontalAlignment.Center;
        tp.TextRange.Text = "Hello, world";

        win.Children.Add(tp);
        win.Show();
    }
}
```

The fact that Microsoft has embraced this new direction for UI development, essentially based on a declarative paradigm, is testimony to the relevance and importance of the researchers who have been advocating this approach for many years, see e.g. (Szekely, 1995).

3.6.6 Adobe MXML and Flex

Adobe recently specified and released a new framework for UI development called Flex, specifically aimed at designing what Adobe calls Rich Internet Applications (RIAs). An RIA is defined as a "web application that has the features and functionality of traditional desktop applications. RIAs typically transfer the processing necessary for the user interface to the web client but keep the bulk of the data (i.e. maintaining the state of the program, the data etc.) back on the application server" (Adobe, 2006).

This term was actually coined in a Macromedia 2002 white paper (Allaire, 2002), although the concept had been gaining importance some years before, and under different names. Among other issues, RIA client technologies should provide an efficient, high-

performance runtime for executing code, content and communications. Among other requirements, it should integrate support for richly formatted text and graphics layout, standard user interface components, allow easy creation of custom user interface components and behaviors, streaming audio and video, and real time communication. The technology supporting the development and deployment of RIAs should therefore integrate content, communications and application interfaces into a common environment. Flash Player, which is now present in 98% of browsers is an example of one of the most successful rich client technologies on the Web, Windows, Unix, PDA's and even cell phones.

The Flex framework builds on the foundation provided by Flash Player 8.5 and ActionScript 3.0 (Adobe Labs, 2006). It adds a rich class library based on ActionScript 3.0 that embodies best practices for building successful RIAs. It also extends the programming paradigm by adding an XML-based language called MXML (which stands for Maximum Experience Markup Language) that provides a declarative way to manage the visual elements of an application. Essentially, the Flex Framework provides the skeleton of the application and allows the designer to:

- Describe an application's user interface from pre-built components by either extending them or creating new ones from scratch;
- Enable predefined interactions, such as draggable columns on a data grid, or hook into some well-defined events to define specialized behaviors;
- Choreograph complex user interface transitions using a flexible effects infrastructure;
- Organize the flow of data through the application's user interface;
- Define the look and feel of the application through a powerful skinning and styling infrastructure.

The developer/designer uses two languages to write Flex applications: MXML and ActionScript (Adobe Labs, 2006). MXML is an XML markup language that you use to lay out user-interface components. One also uses MXML to declaratively define non-visual aspects of an application, such as access to data sources on the server and data bindings between user-interface components and data sources on the server.

ActionScript is an object-oriented programming language used to write programmatic logic for responding to both user-initiated and system-initiated events at runtime.

Like HTML, MXML provides tags that define user interfaces. MXML will seem very familiar to UI developers who have previously worked with HTML. However, MXML is more structured than HTML, and it provides a much richer tag set, like for instance UsiXML. For example, MXML includes tags for visual components such as data grids, trees, tab navigators, accordions, and menus, as well as non-visual components that provide web service connections, data binding, and animation effects. One can also extend MXML with custom components that are referenced as MXML tags.

Figure 3.20 shows a sample application rendered in a web browser window. It contains a List control on the left side of the user interface and a TabNavigator container on the right side.

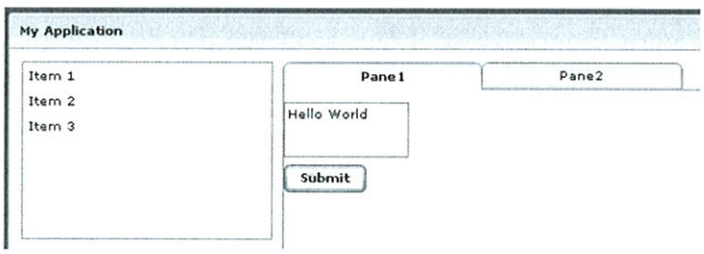


Figure 3.20 A simple example of a UI designed in MXML and rendered in a web browser window.

The following is the code for the sample application illustrated in Figure 3.20. The List control and TabNavigator container are laid out side by side because they are in an HBox container. The controls in the TabNavigator container are laid out from top to bottom because they are in a VBox container.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
       <mx:Panel title="My Application" marginTop="10" marginBottom="10" margin-
Left="10" marginRight="10" >
              <mx:HBox>
                     <!-- List with three items -->
                     <mx:List>
                            <mx:dataProvider>
                                   <mx:Array>
                                          <mx:String>Item 1</mx:String>
                                          <mx:String>Item 2</mx:String>
                                          <mx:String>Item 3</mx:String>
                                   </mx:Array>
```

```
                    </mx:dataProvider>
                </mx:List>
                <!-- First pane of TabNavigator -->
                <mx:TabNavigator borderStyle="solid">
                    <mx:VBox label="Pane1" width="300" height="150" >
                    <mx:TextArea text="Hello World" />
                    <mx:Button label="Submit" />
                    </mx:VBox>
                <!-- Second pane of TabNavigator -->
                    <mx:VBox label="Pane2" width="300" height="150" >
                    <!-- Stock view goes here -->
                    </mx:VBox>
                </mx:TabNavigator>
            </mx:HBox>
        </mx:Panel>
</mx:Application>
```

Adobe also released a set of tools to support this declarative-based UI development process. In particular, Flex Builder provides a state-of-the-art coding and debugging environment; it features intuitive layout, styling, and interaction design tools; and promotes good, maintainable coding practices. What's interesting about Flex Builder is that it tries to congregate both designers and software developers into a single environment.

Developers are presented with flexible visual support for laying out controls by absolute position, relative position, or predefined layout rules (such as vertical, tile, and so forth). Visual design support for "View States" is an innovative feature that enables the user, with very few lines of code, to visually define how the appearance of a control or the entire application changes in response to some event.

Adobe provides a flexible charting extension to Flex. Developers can easily add these data visualization components to any application. The Charting components are dynamically rendered on the client, making it easy to add drill-down, rollover, and other interactivity that makes charts-intensive applications even more insightful than before. The Charting components have a fully exposed API that allows easy customization of the components or the creation of new ones. The charts package is fully integrated into the framework's general infrastructure for effects, styles, data binding, and drag and drop.

3.6.7 *Comparison between Languages*

Comparing the languages described in this section shows that all of them provide multi-platform development capabilities (see Table 3.3). Languages which were born from re-

search activities (such as UsiXML or XIML) are particularly rich in models and are oriented towards model-based development. The XAML and MXML industrial languages are not concerned with models or development methods, instead they focus on interactivity-support features, charting capabilities and data management. All languages produce fully-functional UIs, but XAML and MXML are able to compile and build complete applications. AUIML and XUL are supported only by an interpreter. UsiXML has many tools supporting it, even reverse engineering or sketch-based designers, as described in this chapter. Microsoft is still developing tools to support XAML development; Adobe's MXML uses an IDE and a vector-based interpreter (Flash Player).

Language	*Models*	*Tool Support*	*Orientation*	*Multiplatform?*	*Output*
UsiXML	Abstract UI, Concrete UI, Task, Domain, Mapping, Context	IDE, Reverse engineering, Visual editor, etc.	Model-based development	Yes	Fully-functional UI
AUIML	Dialog, Presentation	Interpreter	Broad	Yes	Fully-functional UI
XUL	Dialog, Presentation	Interpreter	Network applications	Yes	Fully-functional UI
XIML	Task, Domain, User, Platform, Design	Editor, generator, interpreter (UIPilot)	Broad	Yes	Fully-functional UI
XAML	Presentation, Domain	(Under development)	Broad, focused on interactivity, web services, charting and data management.	Yes	Fully-functional application
MXML	Presentation, Domain	Interpreter (Flash player), IDE (Flex Builder)	Broad, focused on interactivity, web services, charting and data management.	Yes	Fully-functional application

Table 3.3 Comparison between the XML-based UI languages described in this section.

3.7 The future of UI Tools

Attesting the importance of our research direction, a recent CHI workshop was entirely devoted to UI tools and their future (Olsen and Klemmer, 2005). This particular workshop aimed at gathering researchers in the field of user interface design tools to identify important themes for the next decade of research.

Bouchet et al. (2005) argued for the importance of designing and building UI tools for recent interaction paradigms such as perceptual UI (Turk et al., 2000), tangible UI (Ishii and Ullmer, 1998), embodied UI (Harrison et al., 1998) and Augmented reality. Adopting a global approach, they propose a component-based approach, called ICARE (Interaction CARE - Complementarity, Assignment, Redundancy and Equivalence), which allows the easy and rapid development of multimodal interfaces.

A much different line of research is followed with end-user programming approaches. In this research direction, the focus is shifted from the interaction designers to the common, not-expert, end-user, and the questions is not how to make tools that empower professional interaction designers to build usable systems, but instead, how to build toolkits that empower millions of end-users to build applications instead of, or as well as, thousands of developers. In particular, (Dey, 2005) is interested in building such a toolkit that supports end-users in building applications for controlling their future smart environments consisting of sensors and actuators. He built 3 different systems for supporting end-users in building context-aware applications: a tangible computing system; a visual programming environment; and, a programming-by-demonstration system.

Myers and Ko argued a similar line of thought, and advocated more natural and open User Interface tools (Myers and Ko, 2005). Their research is stressing some possible directions for the future of user interface design tools. One strategy is to make the tools and their SDKs more usable, effective and understandable by making them more natural. Another is to take advantage of an "open data model" to more easily integrate new components. In addition, programming-by-demonstration techniques and model-based automatic generation still hold much promise (Myers and Ko, 2005).

Puerta (2005) argues for better UI tools through an engineering perspective. He advocates UI tools that are conceived as being part of an engineering process. The tools created

to build interfaces must therefore be devised as tools aimed at engineering activities. Puerta defined four attributes to consider a UI tool as being "natural" (natural in a best-fit manner where the attributes represent the ideal usage of the artifact): *Process Orientation*, not merely in a methodological sense but rather in a procedural or organizational sense; *Interoperability*, which is currently almost inexistent in current UI tools; *Localized Functionality*, UI tools should focus their functionality on a specific aspect of the design process. The rationale for this attribute is simple. Designing a user interface typically involves a number of people with a varied range of skills and roles. To pretend to build software tools that somehow support such a multiplicity of tasks and users is clearly counterproductive. And finally, *Designer Impact*: a UI tool should enhance the limited design skills of non-designers and should transduce the extensive skills of good designers. Puerta calls this "the most ethereal of all the attributes. It is not enough for a UI tool to exhibit all the other three attributes if its use is restricted to very limited groups of people who have very particular talents" (Puerta, 2005).

Looking back at Myers (2000) predictions regarding future issues to be considered in future tools' development, we can verify that some of the tools we described in this chapter are converging towards those predictions. For instance, Myers refers that most toolkits have long assumed that a fixed library of interactive components covered the vast majority of interfaces that were to be built, and that assumption was going to be contradicted (Myers, 2000). This has been happening in industry tools and languages, such as Adobe's MXML and Microsoft's XAML, which provide the developer with extension mechanisms which are easy to employ and adapt. In a related issue, Myers concepts of threshold ("how difficult it is to learn how to use the system") and ceiling ("how much can be done using the system"), along with his prediction that future UI tools should offer low thresholds and high ceilings, is being addressed in the current generation of XML-based languages and corresponding support tools. For instance, MXML has tags for easily designing any kind of chart, and it's fairly easy to add interactivity to charts using this language. The traditional approach would involve creating custom components, learning the language's idiosyncrasies and putting much effort into creating a usable chart component.

3.8 Conclusions and Insights

Future tools will have to be flexible enough to allow the designer think the interaction and not just model it. This kind of support is already offered by some tools that exploit informal input modalities such as gestures, sketch and speech to leverage creativity and exploration in the early stages of development. Nevertheless, more effort will have to be put in this issue, since creativity can occur at any stage of development and current tools predominantly very formal and rigid.

Communication between developers is multifaceted and complex. Furthermore, in UI design, the different backgrounds of a projects' stakeholders often collide, originating several different goals at stake. Current tools support real time cooperation (some even provide awareness and concurrency control mechanisms), but this collaboration mediated by machines is still far from effectively capturing the relationships, ideas and knowledge that happen during real life meetings and discussions.

Future advances in software development will come from teamwork and collaboration. As Grady Booch puts it, "the developer's work is changing from working with an individual tool to developing as a team experience"[2]. Besides providing things such as central-code repositories, where co-developers can work on code together and programmers can freely comment on each other's work, future tools will have to capture and support collective creativity and exchange of ideas in a flexible, engaging way.

Tools will also have to support multiple levels of abstraction, not only in the sense of going from code to model and back, but also in aspects such as system-to-system connectivity, establishing patterns through automated systems and leveraging best practices in general.

Achieving UI tools that provide designers and developers with low thresholds and high ceilings is a complicated task that necessarily involves a deep study regarding UI practices and how design tasks are currently accomplished.

---

[2] Quote from a Charles Babcock article in *Information Week*, Sep 27, 2004.

4 Styles for Workstyles

Framework Foundations

"When your work speaks for itself, don't interrupt"
Henry J. Kaiser, American industrialist (1882-1967)

Being able to capture and describe current work practices in the interaction design field and being able to use and exploit that information in the design of new tools is very important for anyone in the Software Engineering or Human-Computer Interaction fields. As any other conceptual framework, and before presenting the design aspects of the tools developed during this research, it is imperative to contextualize, fundament and define the major concepts as well as the rationale for their definition.

This chapter forms the motivation and foundation of our entire framework. It describes the foundation of the workstyle model, how it was designed, why it is useful and what a designer can use it for. It also presents novel results that contribute to a small but increasing body of empirical knowledge about current tools' usage and modern design and development practices.

This chapter is organized as the following paragraphs describe.

SECTION 4.1 introduces the theme and contextualizes the approach by relating it briefly with the most similar research approaches that are based on the study of work practices.

SECTION 4.2 presents the basic definitions that are used throughout this thesis, and illustrates briefly those definitions recurring to an example of workstyle transition. It also explains why current models don't quite capture this information.

SECTION 4.3 describes in some detail our novel workstyle model, aimed at User-Centered Design (and can also be applied to Usage-centered Design). Each of the dimensions that constitute the model are presented, explained and illustrated with examples. A

set of guidelines (in the form of questions) that can be used to apply the model to tools, notations or styles of work is also provided.

SECTION 4.4 covers the use of the model as a project's stage and effort rough estimator. It describes the relation between a UCD project's phases and the visual characteristics of workstyles plotted in the model's classification space.

SECTION 4.5 shows how the model was used to effectively and easily evaluate and classify some interesting UCD tools (in particular, some tools which were described in the previous chapter).

In SECTION 4.6, we describe and present the results of a survey, performed with the goal of assessing the importance and frequency of workstyle transitions in professional contexts, as well as the tools used throughout those workstyles. It does not make sense to have a powerful, visually-appealing and communication-oriented model about workstyle transitions, if these are not perceived as being significant or if these do not occur in practice. This section describes the foundation of the survey, the demographic data of the respondents, the results obtained for tools and tool usage patterns, and a discussion and analysis of the workstyle-related data obtained.

Finally, in SECTION 4.7, we briefly outline some possibilities about how workstyle modeling could be successfully applied to other problem domains. Questioning whether workstyle modeling (which was already successfully applied to general software development), we present for the first time a lightweight methodology that can be followed in order to incorporate and couple workstyle modeling with any design process. We also exemplify for the case of designing an Arts Center ticket-selling system.

We conclude in SECTION 4.8 where we outline the major lessons learned, reflecting upon the implications of the study's results over designers and tool-makers, as well as setting the stage for presenting concrete examples, under the form of proof-of-concept tools, where designing for workstyle transitions gave rise to new, useful design ideas.

4.1 Introduction

Workstyle modeling (Wu and Graham, 2004) has been proposed as a technique to record the interaction style of a group of collaborators during any software development activity. UCD is an iterative process. This means designers often engage in different workstyles as they iterate towards the final design more often than they do in, e.g. a Waterfall process. This was empirically verified by contrasting the answers to our survey (which we will describe in §4.6). And this is the main reason why we claim that modeling the styles of work can be particularly useful in UCD. It has been widely recognized that current User Interface tools don't support the designers' activities, in particular that UI tools suffer from limited combinations of threshold ("how hard is it to learn?") and ceiling ("how much can be done?") (Myers et al., 2000). There is clearly room for improvement here, since it has been shown that a UI development tool has the potential to influence between 50% and 70% of the application code (Myers et al., 2000). But how could one improve the state of the art in UI tools?

Constantine and Lockwood (1999) described their ideas of "galactic dimensions" as a metaphor change towards fully interconnected and synchronized visual development tools. Traditional CASE tools were based on a metaphor referred to as the "glass drawing board", since they merely represented two-dimensional paper models on the glass surface of a monitor. The "glass galaxy" was then proposed as a multidimensional problem-solving space in which developers could drill down into objects in one dimension, and be taken via software "worm holes" to another. Clicking on a use case could take the developer to its definition in a glossary. Selecting that use case could also show the abstract components that support it, or the concrete widgets for a given realization of that model. Even entries in help files could be linked to the user roles they support, or to the actual code and visual UI controls.

We take this idea further and argue for CASE tools supporting "galactic" dimensions: tools that not only support fast accelerated development through traceability and integration, but also are able to rapidly adjust to any given workstyle in a transparent way. We propose a new workstyle model comprised of eight "galactic" dimensions that we consider as fundamental to supporting the UCD process. Our proposal is new because it is the first

time workstyle modeling is applied to UCD and our model can be used to estimate the stage/effort of development in a graphically intuitive way. Another contribution of this research is that we also show how effective workstyle modeling can be to drive the development of new UCD tools or to choose tool support for a given project phase.

Our workstyle model was based on the identification of the main obstacles to UCD and SE integration (Seffah and Metzker, 2004), current research results and extensive observation of Human-Computer Interaction (HCI) students involved in a UCD project. It incorporates two renamed dimensions from (Wu and Graham, 2004), (Asynchrony and Distribution), and two of the modeling-style dimensions from (Traeetteberg, 2003) (Perspective and Formality). We introduced three more dimensions (Detail, Functionality and Traceability) and unified these dimensions into a common, coherent model, where the axes fall into one of these categories: collaboration style, notation style or tool-usage style.

4.2 Workstyles and Workstyle Transitions

Interactive Systems Design methods, such as (Nunes and Cunha, 2001), often describe users in context by using the concept of actors. Usage-Centered Design (UsageCD) (Constantine and Lockwood, 1999), for example, separates the *actors* of a system from the *roles* they play during the system's usage (see §2.1.4).

Indeed, users adopt several roles during the usage of a system, just like film actors do, but they also switch roles throughout that usage. Although interaction design methods are well conceived to realize systems supporting the roles of usage, few methods provide support for flowing from different contexts/needs of usage.

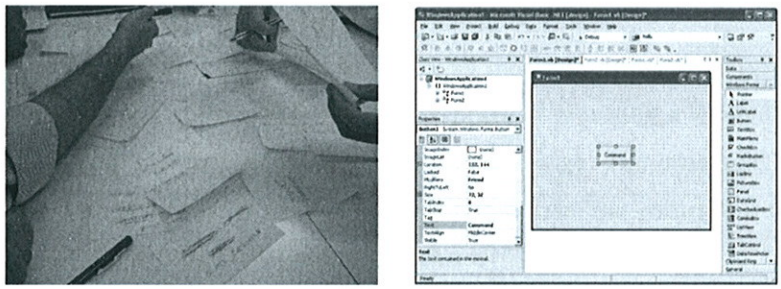


Figure 4.1 A Workstyle Transition: working in groups using low-tech materials for task modeling and clustering (left). After task modeling, each team member is assigned a set of tasks and builds concrete prototypes supporting those tasks using a visual interface builder (right).

Interaction Designers are the users of Design tools themselves, and in this context, we have developed and applied a model for describing the contexts in which they work, by modeling their Workstyles (Campos 2005b, 2005c, 2006a). A *Workstyle* is an informally-defined set of values in *n*-dimensions. These dimensions describe the most important aspects of the way users work in order to achieve their tasks. A workstyle *transition* (or change) is a change in one or more values of a workstyle. A *region* (or *plane*) in a workstyle model is a *set* of workstyles. Systems supporting workstyle regions are systems that can adapt to and support transitions in the users' styles of work. Figure 4.1 shows an example of a workstyle transition in the life of an interaction designer: on the left, a team of developers works together using post-it notes for task clustering in a spatially useful style. After this, the team splits and each designer is assigned a set of tasks and builds a concrete

mock-up of the interface using an interface builder. Each designer transitioned from a low-detail, collaborative, low-tech workstyle to a high-detail, high-tech, individual workstyle.

4.3 A Workstyle Model for User-Centered Design

There are eight continuous axes in our "galactic" workstyle model for UCD, as Figure 4.2 shows. These axes are grouped under three main categories:

- Notation style-related dimensions (Perspective, Formality and Detail),
- Tool usage style-related dimensions (Traceability, Functionality and Stability) and
- Collaboration style-related dimensions (Asynchrony and Distribution), as shown in Figure 4.2.

In this section, we briefly describe each of these dimensions and provide a set of questions that can act as guidelines to apply the model to tools, notations or, in general, styles of work adopted by interaction designers. Throughout this section, the guideline questions will be stated after the description of the corresponding dimension.

It is important to refer that the Workstyle model should be essentially regarded as an informal discussion technique, which has the advantage of providing a vocabulary and a model for analyzing workstyle transitions in interaction design. The main goal is not related to being able to rigorously model every single transition that occurs, but instead to promote discussion, reflection and design inspiration for UI tools.

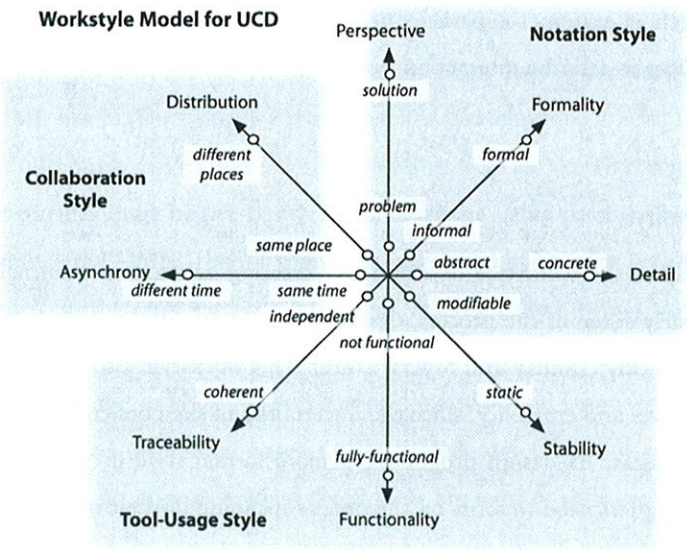


Figure 4.2 A unifying workstyle model for User-Centered Design.

4.3.1 *Perspective.*

This axis plots the perspective, or view, of the artifact being developed. As Figure 4.3 shows, on the lower extreme of this axis (entitled Problem/Requirements perspective) one plots use cases, business logic and goal negotiation and prioritization. As the project gets "green light" to move forward, the designers transition towards the other extreme of this axis, (Solution/Design perspective). Analysis perspective comes in-between, then issues such as modeling, simulation and validation, and finally the finished product.

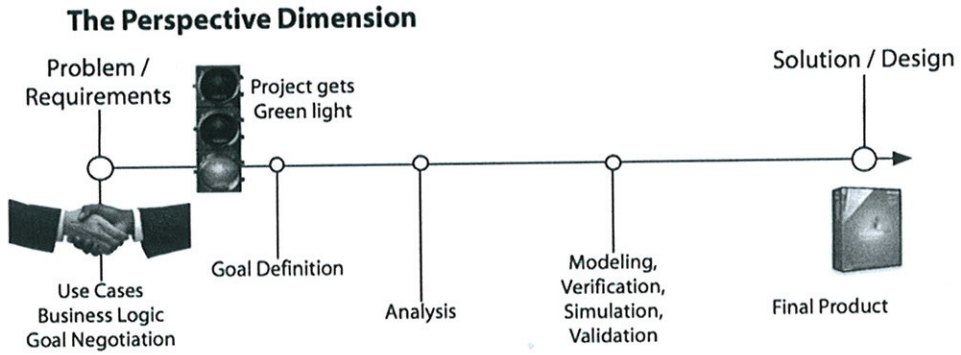


Figure 4.3 Zoom-in over the Perspective Dimension.

Questions: is the notation capable of expressing business goals? Or non-functional requirements such as customer experience requirements? Does it help define the purpose of the system? Does it describe interaction aspects of the system? How close is it to the final product?

4.3.2 *Formality.*

This axis classifies the workstyle of a designer creating artifacts in a formal vs. informal way. In the early stage of the process, designers use rough, ambiguous sketches to freely express ideas quickly (Landay and Myers, 2001). This workstyle also fosters comparison of design alternatives and creativity, since the uncertainty of sketches encourages the exploration of design ideas. As design progresses, a more formal style of work is incrementally adopted, as designers need to focus on the precise meaning of their models. An example of this shift is moving from a whiteboard to a CASE tool.

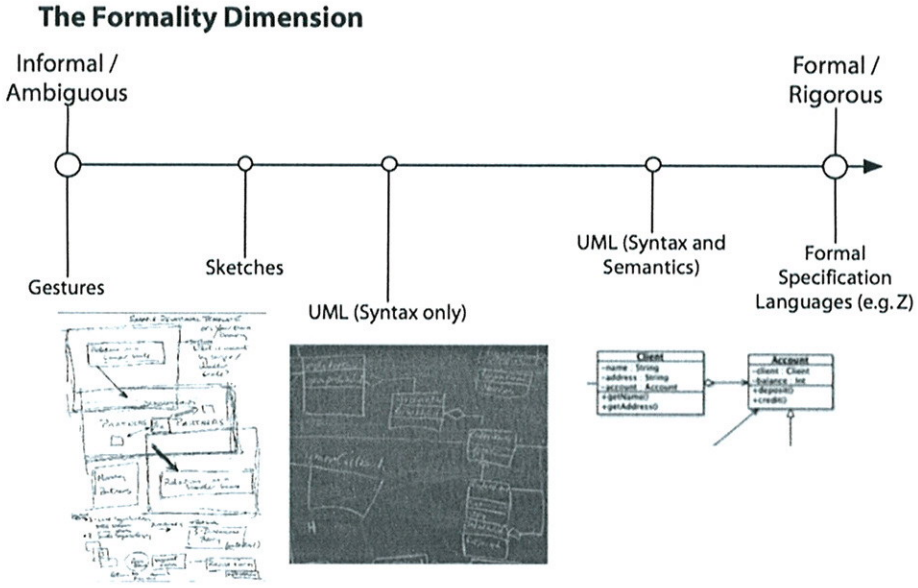


Figure 4.4 The formality dimension.

Figure 4.4 shows the Formality dimension, illustrated with examples which were positioned through the axis. What is the most informal UI description or issue that one could plot? Simple gestures or natural language would be placed at the most informal level. Sketches are also ambiguous and informal (although more meaningful than gestures). Moving along the axis towards the Formal extreme, a language syntactically well-formed, such as the UML has higher formality than sketches. Sketches don't need to obey to any syntactic rules (although this could be done if desirable, e.g. for sketch recognition engines to work, a set of rules must be pre-built). If the language being used is also semantically-sound (for instance the UML), then it should be positioned at an even higher level of Formality. The most formal and rigorous languages, usually called formal specification languages, should occupy the extreme, most formal position. The Z language is a good example and although it is not used in UI specifications, many formal languages for UI descriptions exist such as ICO (a method based on Petri Nets and Object-orientation) (Bastide and Palanque, 1990), an approach where Petri Nets are used to indicate the temporal sequencing of actions and the objects are associated with the system components reacting to

these actions, or TLIM (Tasks, LOTOS, Interactors Modeling), a method which uses task specification to structure the design of the user interface (Paternò and Mezzanotte, 1995).

Questions: how easy is it to define rough ideas? Does the meaning matter? Does the notation force you to use a rigid syntax/semantics?

4.3.3 Detail.

This axis plots the level of detail (or abstraction) the designer is working at. High-level, abstract models facilitate problem solving in organization, navigation and overall structure of the UI, leaving aside the details. On the other hand, realistic (or figurative) prototypes address high-detail design issues (Constantine, 2003). Disciplined designers tend to assume a workstyle that goes from higher-level abstract representations towards more realistic and detailed representations as the process evolves (Constantine, 2003). This means that in our model, a disciplined workflow would start out at the lowest level of abstraction and then smoothly move on to the highest level of detail (a concrete, visually fine-tuned UI prototype).

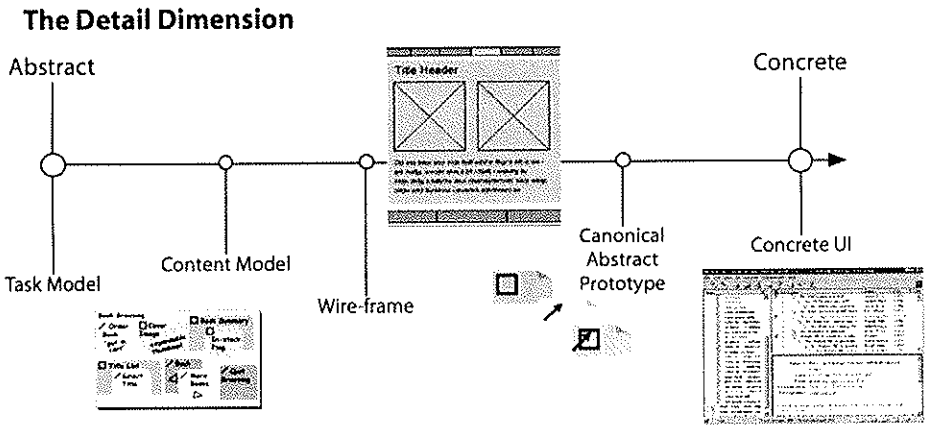


Figure 4.5 Some examples of UI-related artifacts at different levels of detail.

The Detail dimension is illustrated in Figure 4.5, where we show how UI artifacts are created at very different levels of detail. On the lower extreme of this axis (labeled "Abstract") one could plot Task Models or User Profiles. A UI content inventory would still be abstract but should have a higher value because it's closer to the concrete, detailed solution

than a task model is. A wire-frame has more detail than a content model, since it contains detail about spatial layout of the UI elements (a content inventory is just a list of elements). Moving forward in the detail axis, a Canonical Abstract Prototype has more detail than a wire-frame since it also models the interactive function of the elements using a catalog of possible interactive functions. At the highest level of detail (labeled "Concrete"), one could position concrete UI prototypes, detailed and fine-tuned either at visual or behavior levels.

Questions: can you abstract irrelevant details using the notation? Can you think about navigation and structure of the overall interaction using the notation? Can you incrementally add enough detail?

4.3.4 Stability.

This dimension describes how difficult/frequent it is to modify any aspect of the artifact(s) being developed. Note that while Perspective, Formality and Detail were Notation-style related, the next three dimensions in our model (Stability, Functionality and Traceability) are tool usage-style related. This means they concern what you can do with a given tool, and this includes any kind of tool, even low tech tools such as paper and pencil.

Regarding Stability, UI artifacts can be easy to modify, like a digital model created in e.g. VISIO (plotted in the lowest extreme of the axis, labeled "Modifiable"), or extremely difficult to change and edit, like paper and pen storyboards (plotted at the highest level of Stability). This is illustrated in Figure 4.6.

A content inventory of the UI modeled in a UML tool is highly modifiable because it is easy to change names, positioning, size and other aspects of the elements. This is opposed to drawing a model of the UI with pen and paper, since changes are harder to accomplish. Brainstorming, for instance, is a very unstable workstyle because changes are very frequent. High values in this axis indicate less frequent or less significant changes.

Questions: How easy is it to modify previously created artifacts using the tool? How frequently do you make those changes? Are there particular changes difficult to accomplish with the tool?

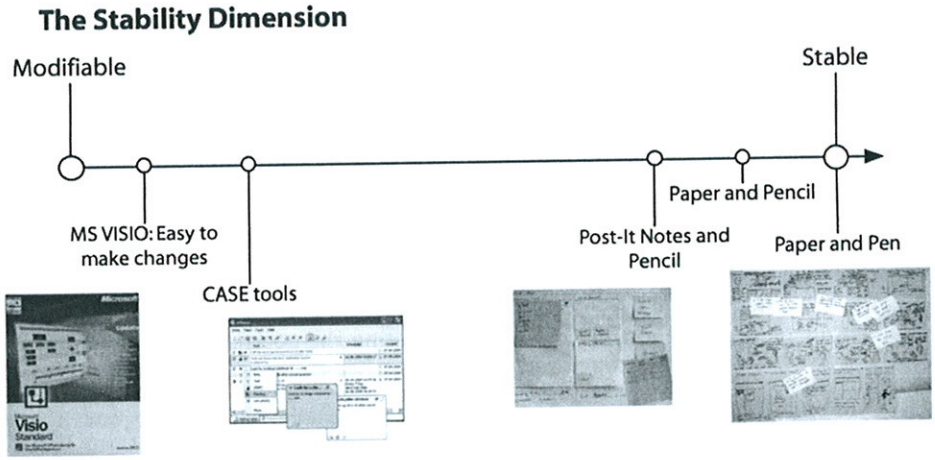


Figure 4.6 UI artifacts can be easy to modify, like a digital model created in e.g. VISIO, or extremely difficult to change and edit, like paper and pen storyboards.

4.3.5 Traceability.

This is dimension describes if the elements of the artifact being developed are consistent and interconnected (thus being highly traceable) or if they are completely unrelated and independent. As an example, developers might adopt a workstyle in which they choose to keep links from task cases steps and the concrete UI widgets that implement those task steps. In this case, it is possible to trace a task step to the concrete widget and to trace a widget to the task step it implements.

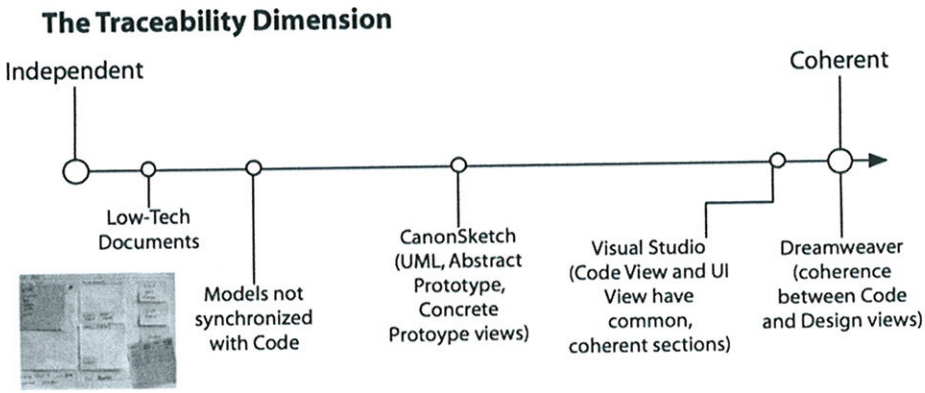


Figure 4.7 The traceability axis depicts whether models are synchronized and kept coherent, or if they are kept independent of each other.

This dimension is closely related to stability and the number of artifacts produced during a project. As they increase, traceability becomes more important. However, during the early stages of design, when the designer is creatively trying to consider the largest amount of possible designs, traceability is not an issue.

To better describe the Traceability dimension, consider Figure 4.7. The lowest extreme of the Traceability dimension is labeled "Independent". This means there are no traceability concerns when using the tool. For instance, when modeling with the help of low tech tools such as post-it notes, it is very difficult to achieve traceability (one could circle a number for related items but it's still difficult). On the other extreme (labeled "Coherent"), consider tools like MS Visio or Dreamweaver. In these tools, both Code and Design views are always kept in a coherent state and are highly traceable: for instance double-clicking on a UI widget can take the user to the corresponding code that implements that widget's behavior.

Questions: Are you using the tool to maintain interconnections between model elements? How important is it to navigate through your model? Does the tool maintain several different views in a synchronized way (e.g. design view and code view)?

4.3.6 Functionality.

This is dimension represents how much functionality is being addressed by a given tool. There is a barrier between software engineers and usability professionals regarding this matter. On the one hand, software engineers are engaged into building reliable, functional systems, leaving user-friendliness to the usability specialists. On the other hand, usability and interaction designers first design and test the interface with end-users, leaving implementation to software engineers, regarded as functionality builders. Those two processes should not be separated (Seffah and Metzker, 2004) and considering this dimension will help overcome that barrier. This dimension is also important because designers combine visual design (presentation issues) with dialog design (behavior issues).

Figure 4.8 shows some examples of UI design outputs with different levels of Functionality: a static UI mock-up used for paper prototyping (Snyder, 2003) would be placed with a low level of Functionality, of course. A Storyboard would have a slightly higher value of Functionality: although it doesn't "work", it describes more precisely how the prototype

will work. A fully-working, completely specified and tested UI prototype should be placed at the highest end of the dimension (labeled "Fully-Functional").

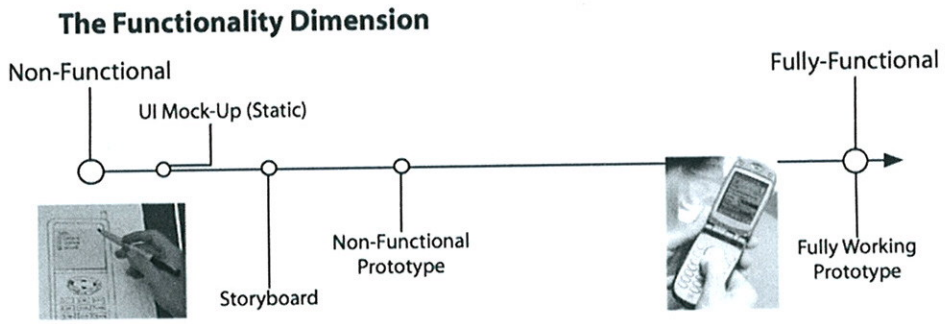


Figure 4.8 The functionality axis depicts the level of functionality present in a UI artifact.

Questions: How much functionality, behavior and dynamics can you add to your prototypes using the tool? How easy is it to test the interaction by using the tool?

4.3.7 Asynchrony.

The last two dimensions are Collaboration-style related. The Asynchrony axis refers to the collaboration style that designers assume the following way: they can make changes to the work being developed at the same time (a synchronous workstyle) or they can work at different times (engaging in an asynchronous workstyle) (Wu and Graham, 2004). The higher the value in this axis, the more asynchronous is the workstyle.

Building tools that support UI-related work at the same time involves dealing with the maintenance of a shared model's coherence (if changes are being made remotely). Building tools that support work at different times involves finding usable ways to contribute, trace and manage changes to the artifacts. Commonly used tools for synchronous workstyles include whiteboards (or the smarter, much more expensive, equivalent boards called smartboards). Commonly used tools for working asynchronously include e-mail, CVS systems, centralized repositories and blogs (see Figure 4.9).

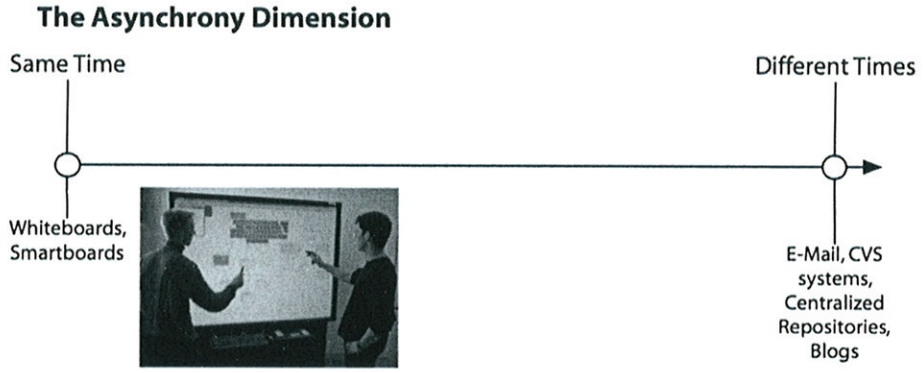


Figure 4.9 Asynchrony is the dimension that plots whether designers work at same time or at different times.

Questions: do the team members change artifacts at the same time? Or do they make changes at different times? How frequently?

4.3.8 Distribution

This last dimension describes whether work is being conducted at the same physical location or at geographically distant locations. As shown in Figure 4.10, commonly used tools to co-located workstyles include whiteboards or sheets of paper on the top of tables. Use of webcams and video messaging is useful for remote collaboration workstyles.

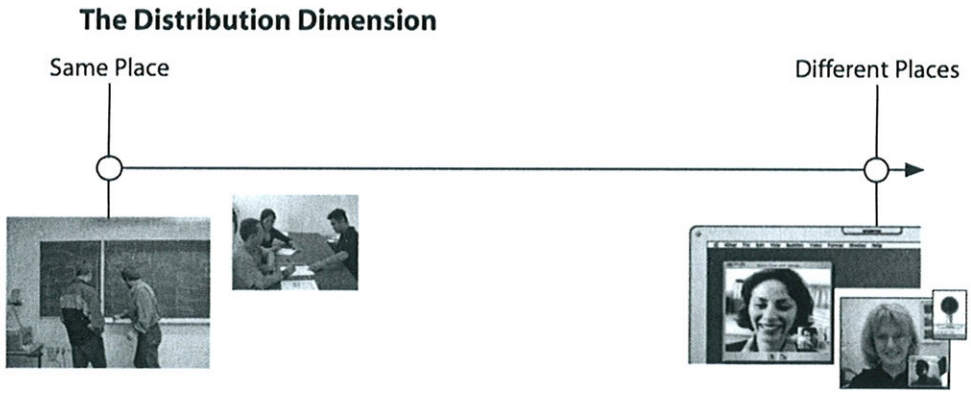


Figure 4.10 Distribution: designers work at the same place but they can also work at different places.

There is a much significant difficulty in building tools that support remote collaboration (working at different places) than co-located collaboration.

Questions: how far are the team members collaborating? Are they in the same building? Or are they in a different continent, or scattered through a country?

These dimensions can be effectively used to assess a given workstyle adopted by an interaction designer or a team. A single workstyle is plotted as a line (a point in the eight-dimensional space) whereas regions (or planes) represent sets of workstyles.

4.3.9 Practical, Real-world Workstyle Examples

There are many real-world situations where workstyle transitions come into play, especially at turbulent development environments, in professional settings, but even in quieter contexts, such as UCD classes or lectures, it is possible to observe people engaging and transitioning between workstyles. Some transitions occur very frequently, others are not so frequent but are very costly. By cost, we mean difficult to perform, either because the effort required to achieve them is considerable, or because the time required to the transition is high. In this subsection, we will demonstrate how the Workstyle Model can be used to rapidly and visually characterize workstyle transitions.

All figures in this subsection plot the initial workstyles as a thick, continuous line, and the final workstyles are plotted as a thin, dashed line.

The first example was already described in section 4.2 on page 121: it exemplifies a transition from low-detail, low-tech, collaborative workstyle (working in groups to model and cluster sets of tasks to be supported by the system) into a high-detail, high-tech, individual workstyle (individually building a concrete screen for each cluster of tasks using a visual interface builder).

Figure 4.11 shows how to plot this transition using the Workstyle Model: the initial workstyle presents a medium value for perspective (task modeling is essentially analytic), and it is informal, low detailed, performed in a stage with intermediate stability, without functionality, no interest in maintaining traceability between artifacts (which would be difficult to achieve anyway since designers were using post-it notes), performed at the same time and place (anybody can just add/remove a card over the shared table). The grey ar-

rows show the direction of the transition: when switching to Visual Studio in order to build a concrete prototype, designers transitioned to a design/solution level (increasing the value in the Perspective axis), formal and highly-detailed, more stable, adding full functionality, maintaining traceability (in this case, maintaining coherence between code and UI design), and performing all this at different places and at different times (high values in the Asynchrony and Distribution axes).

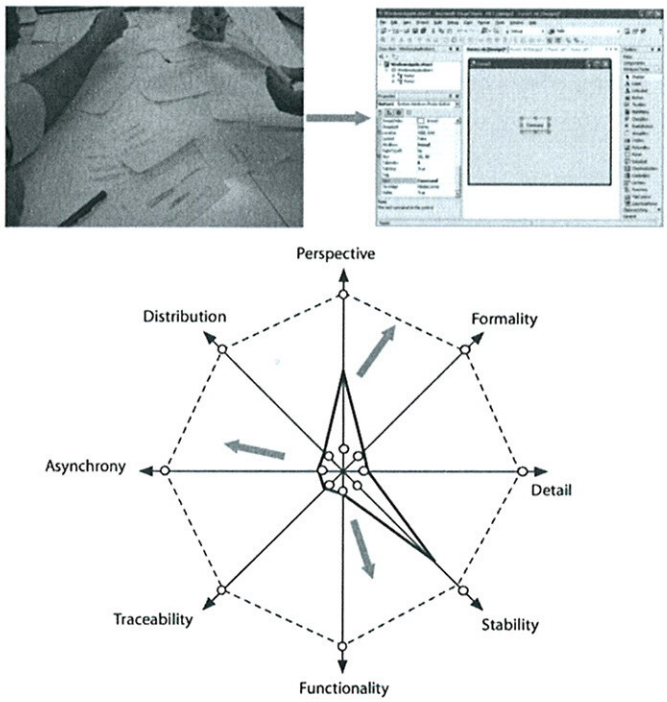


Figure 4.11 An example of how the Workstyle Model presented in this thesis can be used to classify and depict a workstyle transition. In this case, transitioning from a low-detail, low-formality collaborative workstyle into a highly-detailed, highly-formal solution-level and individual workstyle.

As an example of a workstyle transition associated with high cost, consider the example in Figure 4.12: moving from collaboratively sketching a UML class diagram in a blackboard into translating that same model into a digital, semantically-sound model using a UML tool. This transition is clearly difficult to perform since (i) it requires a manual effort to re-create the entire model into the tool, and (ii) the initial model is a rough sketch,

therefore it is ambiguous which might cause the designer to recapitulate the context and the exact information that was defined in the collaborative model sketching.

Figure 4.12 also depicts the transition in terms of the Workstyle Model: the notation-style Perspective and Detail values remain unchanged during the transition, because the notation is the same: UML class diagrams are at the analysis perspective, at intermediate level of detail. However, if the UML digital tool builds a semantically-sound version of the model (and validates it), the value for formality increases.

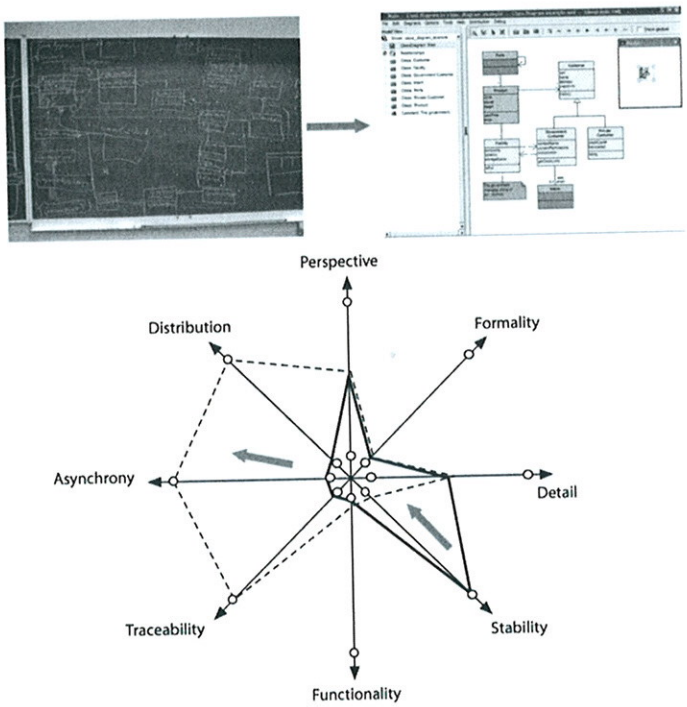


Figure 4.12 A transition with high cost (moving from collaboratively sketching a UML Class diagram in a blackboard into a digital version of that same mode using a UML-based tool), and the corresponding representation using the Workstyle Model.

The value for Stability also changes, since it is clearly easier to perform changes on a digital model rather than using chalk on a blackboard. Functionality is kept the same (although some UML tools are able to generate code from e.g. Activity Diagrams). Traceability increases when moving to a digital tool, because one can maintain digital connections between the relevant model elements. If the tool does not allow designers to continue

modeling at the same time and place, the values for the Collaboration-style dimensions also increase, as the Figure shows.

When using the Workstyle Model, one can also remove the axes that don't add any information to describing the workstyle transition. For instance, if there are no differences in Collaboration-style, one can simply remove them and get a simplified pictorial description of a given transition (see Figure 4.13). It shows an example of a workstyle transition with high frequency: moving from a concrete, fully-designed UI prototype back into the use cases and content model to see if every task and every abstract model in the UI content inventory are well supported in the concrete UI.

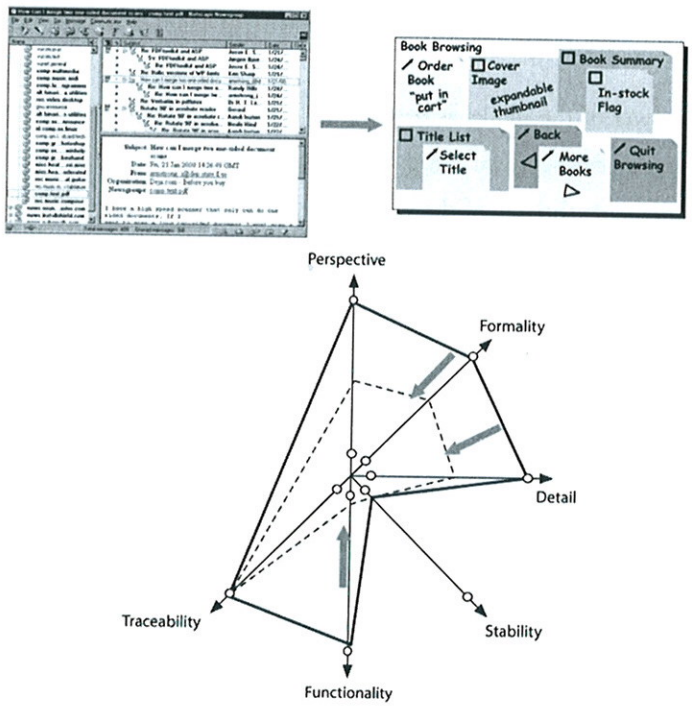


Figure 4.13 A workstyle transition with high frequency: designing a concrete UI prototype and going back to use cases and/or user models to see if they match and/or are well supported by the proposed solution. This transition implies a change in perspective, formality, detail and functionality. One can remove the axes that don't quite influence the description of the transition (in this case the collaboration-style axes were removed from the picture).

As we can see from the Workstyle Model depicted in Figure 4.13, this transition implies a change in perspective (from the highest, solution level to a lower one), formality, detail (the designer transitions into a lower-detail notation), and functionality (from fully-

functional to non-functional). The collaboration-style dimensions weren't relevant so they were simply removed (Traceability and Stability also remain unchanged so they could have been removed as well).

All these examples are intended to present Workstyle Modeling as an informal technique for reflecting upon the transitions that are performed. Being an informal discussion tool, the point here is not to rigorously model all the possible transitions at a very exact level, but instead to promote discussion and a common language/method to depict, communicate and analyze transitions in the life of interaction designers. Thus, if we were to apply the model to itself, the formality value would be low (it is essentially an informal tool) as well as the values detail and stability (because it's easy to plot changes using the model).

The visual power of the model is better described in the following section: it gives an idea of the stage of development designers are engaged in, through a visual and easy to perceive measurement: the size of the perimeter plotted. The larger the size, the closer to the end of the product development.

4.4 Using the model to assess a UCD Project's Stage and Effort

In general terms, it seems reasonable to say that as the process evolves, designers tend to assume a workstyle that spreads them away from the center of our model.

Under the perspective axis, for example, it is clear that as time goes by, developers move from the domain/problem level towards the solution/design space. In terms of formality, they start out with informal, ambiguous sketches and move to formal languages later on, when coding increases and functionality becomes more important. Under the detail dimension, as we have already seen, skilled designers (and even non-skilled ones) tend to go from higher-level abstract representations towards more realistic and detailed representations as the process evolves.

Also, as the deadline approaches, prototype functionality is added (and is needed for user testing and customer delivery). In an initial phase, designers don't spend much effort on functionality: it is more important to rapidly compare design alternatives. In addition, and also as time goes by, ideas start to solidify and changes become smoother, rather than significant (thus increasing stability). Under the traceability axis, the number of artifacts increases, and so does the number of inter-connections between them, which motivates the need for increased traceability. This also happens because developers are not interested in throwing away the models (as in brainstorming) but rather in keeping all of the models created so far in a coherent, traceable state.

Under a collaboration style perspective, the transition may not be so straightforward. Nevertheless, as development tasks get larger and work allocation is made, there is a tendency to work asynchronously and at different places (thus increasing the workstyle value on the Asynchrony and Distribution dimensions), because developers feel the need to focus and split work.

As we will see, one of the advantages of our model is the fact that it graphically conveys implicit information regarding the temporal stage of development (again, in general terms).

Although our analysis is based on a specific development process (the Wisdom process framework), we believe the results could be similar for any other processes used. Therefore,

part of a possible future work would include applying the workstyle model to other frameworks and methods.

In the Wisdom process framework, the earliest phase is the Inception phase. During inception, the most significant activities performed are related to project comprehension (understanding a project's complexity and scope), understanding the system context, and modeling user profiles and tasks. As illustrated in Figure 4.14, the workstyles prevailing during the inception phase are generally low-valued in every dimension. Designers work at low levels of detail and formality, changing issues frequently, not worrying about functionality or traceability, and usually working collaboratively at the same time and place. The perspective level is also low (designers interiorizing a project work at problem perspective first).

Figure 4.14 also illustrates that during this phase, transitions are not very frequent, thus the effort put is relatively low (as compared to the following phases as we will see) and the workstyle model's perimeter size also reflects the stage of the development. The smaller the size of the perimeter plotting the workstyle line, the earlier the phase in the project. We will see, through similar analysis and discussion of the next phases, how it grows according to the project's stage of development.

Inception Phase

Activities:
- Interiorize Project
- Understand system context
- User Profiling
- Task Modeling

Effort:
LOW

Project Stage:
EARLY

Transitions' Frequency and Cost:
LOW

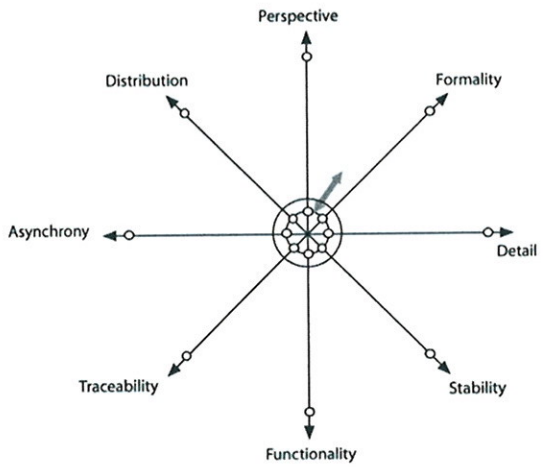


Figure 4.14 Inception Phase and the corresponding average transitions.

The next phase, Elaboration, is based around task modeling, internal system analysis and designing the interface architecture. As Figure 4.15 plots, the workstyles involved in

the activities present during the Elaboration phase are farther away from the center of the model. Task analysis, internal system analysis and initial designs of the UI's architecture force designers to adopt workstyles with slightly higher values than during the Inception phase. It is clear that this phase occurs at an intermediate stage of development, and since task modeling and the other activities involved in this phase usually cause designers to go back to the previous issues like interiorizing the project's context, one can reasonably assume that the frequency of workstyle transitions increases.

Elaboration Phase

Activities:
- Task Modeling
- Internal System Analysis
- Interface Architecture Design

Effort:
MEDIUM

Project Stage:
INTERMEDIATE

Transitions' Frequency and Cost:
MEDIUM

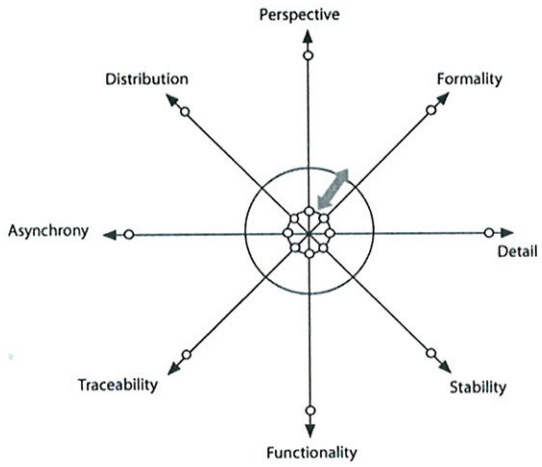


Figure 4.15 Elaboration Phase and the corresponding average transitions.

Figure 4.16 shows what happens in the Construction phase. The activities that are more characteristic of this phase are internal system design, user interface design and prototyping implementation. This implies an increase on the values in the Traceability and Functionality axes. Also, and since prototyping (an activity solution/design-oriented) usually shifts the focus of attention towards increased detail and formality, the values in Perspective, Formality and Detail also increase.

The level of effort during Construction is high. And so are the workstyle transitions' frequencies (and costs). Once again, the perimeter of the plotted lines is larger, reflecting that we are on a later-intermediate phase of the project.

Construction Phase

Activities:
- Internal System Design
- User Interface Design
- Prototyping / Implementation

Effort:
HIGH

Project Stage:
INTERMEDIATE

Transitions' Frequency and Cost:
HIGH

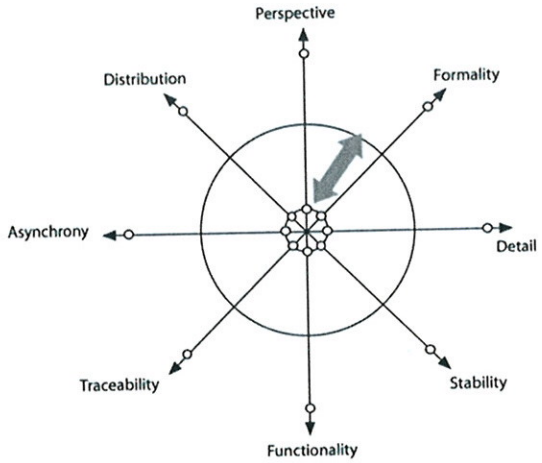


Figure 4.16 Construction Phase and the corresponding average transitions.

Finally, in the Transition phase, illustrated in Figure 4.17, the effort is at its highest, the project's at its final stage, and workstyle transitions are more frequent and costly (as defined in §4.3.9) than never. The associated activities (prototyping/implementation and evaluation/test) are usually done in workstyles with high values, spread away from the center of the model.

Transition Phase

Activities:
- Prototyping / Implementation
- Evaluation / Test

Effort:
VERY HIGH

Project Stage:
FINAL

Transitions' Frequency and Cost:
VERY HIGH

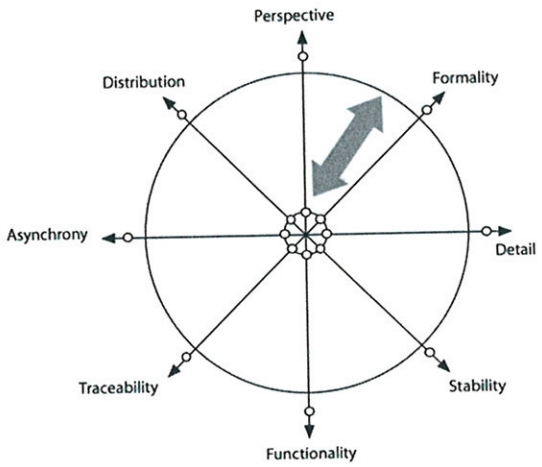


Figure 4.17 Transition Phase and the corresponding average transitions.

To conclude and review the discussion presented here, Figure 4.18 shows the workstyle model plotted for the several phases of a UCD process (in this case the WISDOM process

framework (Nunes and Cunha, 2001) which we described in §2.1.9, page 40). This is a rough modeling of the styles of work and how they vary according to the activity being performed. In the inception phase, all workstyle dimension values are low: developers think informally in terms of requirements, at the same time and place, without functionality issues in mind, doing many changes to compare alternatives. As they move to elaboration and construction, they start to adopt a workstyle with higher values in all dimensions (although some more than other).

The grey area shows the effort along the time, for each activity. The circles show the workstyle adopted by developers along the time as well. When the effort is higher, workstyle transitions become more frequent (the circles' thickness plots the frequency of transitions, the thicker the circle, the more frequently the transitions occur).

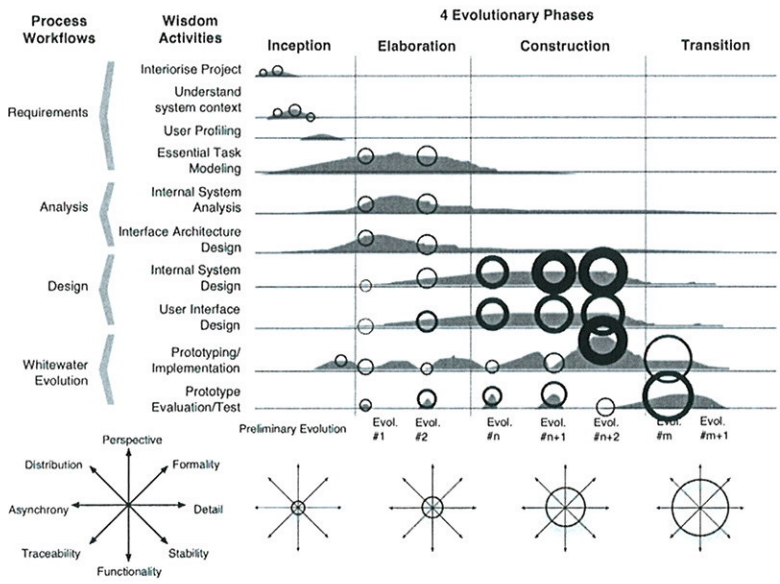


Figure 4.18 The workstyle model plotted along the different phases of a UCD process.

In practice, this evolution is never translated into a perfect circle, as different projects have different goals and needs. For instance, the larger the organization, the greater the formality and location dispersion. However, if we think in terms of workstyle iterations, as design evolves, there is a general tendency to move away from the center of our model. Also, if regions (instead of lines) become plotted in our model, we have an indication that

iterations and workstyle transitions become more frequent, which accounts for higher project effort. Consequently, by checking the plotted regions' size, one can estimate how much effort is being put by a team of UCD developers. The power of our model (over other models such as (Green and Petre, 1996; Traeetteberg, 2003; Wu and Graham, 2004)) is the fact that it can be used to assess the level of development in a graphically intuitive way. An image can convey more information than words or numbers and models should make use of them. In the next Section, we will show how we used the model to effectively and easily evaluate and classify some interesting UCD tools.

4.5 Using the model to evaluate and select UCD tools

Besides the relation with a UCD project's effort and stage, our "galactic" model can also be used to identify adequate UCD tools for a given project phase. In this section, we will consider the value of our Workstyle model not only for choosing and classifying UCD tools, but also – and perhaps more importantly – to spot mismatches between a tool and the intended workstyles that should be supported by the tool. This is a new form of evaluating a tool which is independent of usability evaluation techniques: the purpose is not to measure the usability of the tool, but the adequacy of the tool to the intended workstyle or set of workstyles.

Let us consider, for instance, brainstorming. In this workstyle, all values of our model are very low: problem focused, informal, low detail, unstable (many ideas and many frequent changes), no functionality at all, no need for traceability and people collaborating at same place and at the same time.

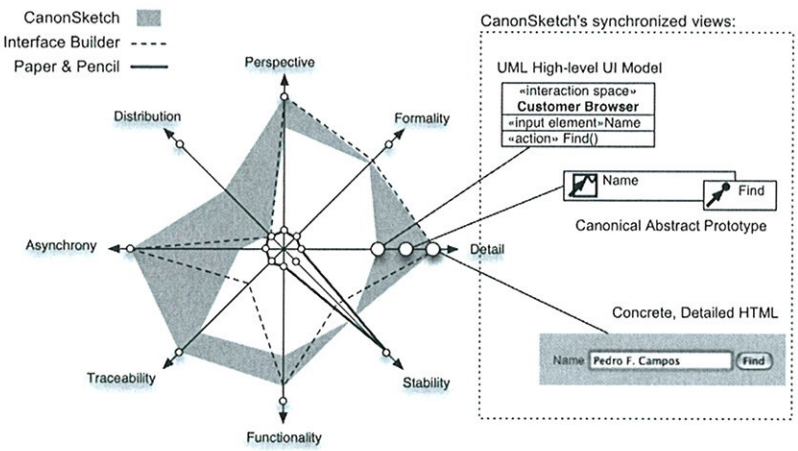


Figure 4.19 CanonSketch, Interface Builder and Paper & Pencil under the galactic model.

Figure 4.19 shows the workstyle model of Paper & Pencil as a thick line. Through our model, we can see that it is a tool almost perfect for brainstorming. However, changes are hard to accomplish using paper/pencil, so the value for stability is high. By looking at the model, we get an indication of the mismatch point between the tool and the desired work-

style. The ideal tool for brainstorming would also need to support fast changes to artifacts, as computer tools do. This is an example of how to use the model in order to spot a tool-workstyle mismatch.

Figure 4.19 also shows how our model was used to drive the development of a new user-centered tool for designing UI's. CanonSketch (Campos and Nunes, 2004a; Campos and Nunes, 2004b; Constantine and Campos, 2005) is a tool that supports multiple levels of detail by providing the designer with several views: UML model of the UI and domain, Canonical Abstract Prototype (Constantine, 2003) and HTML concrete UI (as the right side of Figure 4.19 exemplifies). The first two views are synchronized and the UML semantic model is used to support traceability. There is also a collaborative version of this tool in which designers can work at the same time on the model and at different places. However, support for distribution is still limited (for instance, there are no awareness mechanisms). Therefore, CanonSketch supports a region in our model, as illustrated in Figure 4.19. In this way, the tool seamlessly supports designers while switching from high-level abstract views of the UI and low-level concrete realizations (Campos and Nunes, 2004a). CanonSketch has been tested under a laboratorial setting and has lead to promising results (for a detailed description about CanonSketch's evaluation please refer to CHAPTER 6 in this thesis). By contrast, a conventional Interface Builder only supports a line in the workstyle model (the dashed line in Figure 4.19): UI's are designed at the highest level of detail, at a solution perspective and designers can only work at different times and at the same place (unless the interface builder provides support for collaboration).

Figure 4.20 illustrates the application of the model to two software modeling tools: ArgoUML and IdeogramicUML. The former is a well-known open-source UML tool and the latter is a commercial tool based on a research project about collaborative software design. ArgoUML is described in SECTION 3.2.2 of this thesis, and IdeogramicUML is also presented and discussed in SECTION 3.4.4. ArgoUML has a full-semantic model that allows, for instance, reverse engineering and code generation. IdeogramicUML only supports the syntax of the UML. Thus, ArgoUML is more formal than IdeogramicUML. They both cover part of the perspective, abstraction and modifiability axis; and can even generate some partially functional code. Traceability exists to some extent, since some views are interconnected automatically and there is something like a model navigator in both tools.

More differences come in the collaboration-style dimensions. IdeogramicUML uses a sketch recognition language and can be effectively used in electronic whiteboards. There is also a distributed version with awareness mechanisms built in. This way this tool covers a larger region of the model (see Figure 4.20) than ArgoUML.

Figure 4.20 also compares the galactic workstyle values for a visual interface builder (VisualBasic) against the popular diagramming tool MS Visio. Visual interface builders became very popular because they support a workstyle in which a functional, concrete prototype can be easily created, thus reducing iteration times. Like ArgoUML, there is no support for collaboration. These tools are limited to a single workstyle (plotted as a single line), so there is no support for abstraction or requirements definition, which is the reason why so many interfaces are quickly, but poorly designed. On the other hand, Visio supports a wide range of detail, perspective and formality levels, and even though it doesn't create functional prototypes, its flexibility in terms of notation quickly became key to its success in rapid prototyping.

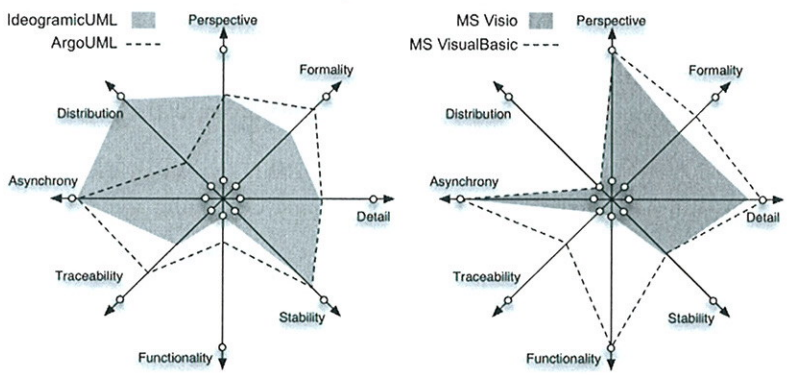


Figure 4.20 ArgoUML, IdeogramicUML, VisualBasic and Visio, under the galactic model.

These examples show how it is possible to find adequate UCD tool support by applying our model in an easy and intuitive way. It also shows how we can compare and analyze the trade-offs between the dimensions.

More importantly, our model can be used to drive the design of UCD tools, as we have already done with the CanonSketch project. The ideal UCD tool should support the whole

space of our "galactic" model as well as shifts between any given workstyle. This could leverage the iterative nature of the UCD process.

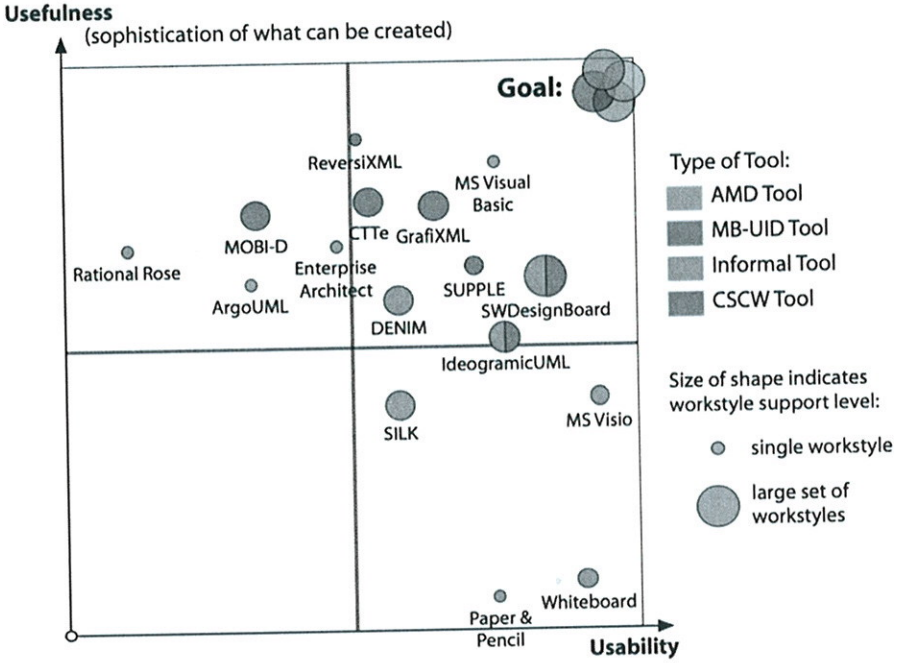


Figure 4.21 A four-dimensional, informal analysis of some of the current UCD tools (most of them surveyed in Chapter 3, State of the Art).

Figure 4.21 summarizes the results of analyzing and classifying some of the surveyed tools in CHAPTER 3, using a four-dimensional plot, where the x and y axes represent the levels of usability vs. usefulness found for a given tool, and where color indicates the type of tool, the size of the shape indicates how well does the tool support regions in our workstyle space.

Typically, sophisticated AMD tools (such as Rational Rose) tend to present much functionality at the cost of poor workstyle support and a high learning curve that makes them very hard to use tools (there are even companies dedicated to provide training in using these tools).

On the other extreme of this classification framework, low-tech tools such as paper & pencil or whiteboards are very usable but they are unable to produce sophisticated artifacts. Informal tools such as MS Visio, DENIM (Landay and Myers, 2001) or Ideo-

gramicUML (Damm et al., 2000) are very usable and supporting a considerable surface in the workstyle space. However, they don't produce full semantic models or support tasks such as formal scenarios execution and simulation.

The goal is, therefore, to achieve highly useful and highly usable killer-apps for any class of AMD, MB-UID, Informal or CSCW tools, while maintaining support for many styles of work as well as transitions between those styles.

It does not make sense to have a powerful, visually-appealing and communication-oriented model about workstyle transitions, if these are not perceived as being significant or if these do not occur in practice. To better understand transitions and tools, during the life of interaction designers, in the next Section we will describe and present the results from a study, based on a survey, performed with the goal of assessing the importance and frequency of workstyle transitions in professional contexts, as well as the tools used throughout those workstyles.

4.6 A Survey on UI Workstyles and Tools

The fact that it is possible to describe a model for capturing workstyle transitions does not mean that transitions actually occur. We wanted to know whether workstyle transitions were effectively regarded as frequent and/or difficult by professionals engaged in any kind of UI-related activity. We were also interested in gathering information about the practitioners' tools of the trade in a valid, reliable and significant way. This section describes some answers we obtained for the two main research questions that we posed:

(a) Are workstyle transitions really perceived as frequent and/or costly, in a professional setting? And if so, which transitions present more difficulty and which are more frequent?

(b) What tools are currently most used to perform UI-related activities?

Although there are many studies devoted to analyzing general software development practices, the literature for qualitatively studying UI-related work practices in software development is relatively rare. We try to increase this small but growing body of knowledge by arguing the importance of supporting workstyle transitions in UI practices and presenting (i) a survey of 370 professional practitioners' answers to their current tools' usage and workstyle transitions and (ii) concrete examples in the form of design tools that try to support the most important transitions. Our study was specifically aimed at interaction design and user interface activities, but since these activities can also be performed by non-interaction designers, we also gathered a significant amount of responses (170) from programmers, system analysts, and project managers.

The entire planning, design and realization of the survey was inspired and based on Dillman's *Mail and Internet Surveys* reference book (Dillman, 1999). We also conducted follow-up interviews to corroborate some of the conclusions presented, and we will also present (in CHAPTER 5) novel ways to support workstyle transitions, that were built on these and previous results.

In this section, we will describe the demographic data of the respondents, the results obtained for tools and tool usage patterns and the workstyle transitions-related data from our survey.

4.6.1 Foundation of the Survey

Dillman, one of the most well-known researchers in the field of survey-based research, describes the use of several implementation elements to achieve high response rates. In 1978, he developed a general method of implementation, known as the Total Design Method, which is known to achieve high response rates. Since then, Dillman expanded this design and re-named it *The Tailored Design Method* (Dillman, 1999). This method is very useful and gives explicit recommendations that we tried to follow, such as designing a respondent-friendly questionnaire, shortening the questionnaire (asking fewer questions to reduce respondent burden) and creating questions that the respondent will find interesting to answer.

However, and since surveys are one-sided (Ferré et al., 2001), we also conducted formal and informal usability studies on workstyle transitions (see CHAPTER 6), as well as follow-up surveys with some of the respondents in order to gather ideas and corroborate some conclusions.

Seffah and Kline (2002) showed that there is a gap between how programs are represented and manipulated in tools and the actual experiences of software developers. They measured in a quantitative way the developers' experiences using heuristic and psychometric evaluation. However, the UI-related issues are not specifically studied and those issues have been our main concern in this research, which is also related to the problem of integrating usability and software engineering (Seffah and Metzker, 2004). Another research approach similar to our strategy is recording the tools' usage events (like copying text, searching the editor, studying search results etc.) and then analyzing the data to find frequencies of patterns (Singer et al., 1997).

This was the starting point of our investigation towards studying the interaction design aspects of software development, in terms of tools and workstyle transitions. Based on previous research findings (Campos and Nunes 2005b, 2005c), and on the reference book by Dillman (1999), we conducted a survey for studying the tools and workstyles of practitioners performing UI-related activities.

4.6.2 *Respondents' Demographic Characteristics*

We disseminated the questionnaire to two major interaction design mailing lists: CHI-WEB and the IxDA (Interaction Design Association), as well as many industry contacts. In less than two weeks, we collected more than 225 valid responses. Appendix B, "A Study on Workstyles and Tools for Interaction Design", provides the full text of the distributed questionnaire.

But since software engineers (system analysts, programmers) also engage in UI-related activities, we also surveyed two software development mailing lists: IS-World and Flex-Coders. In this second dissemination phase, we gathered another 145 responses (more than 90% from System Analysts and Programmers), thus achieving a total sample size of 370 respondents. We found that the results were similar for both Interaction Designers and System Analysts/Programmers, which wasn't surprising since the activity was the same (UI-related). In the initial part of the questionnaire, participants were asked questions about their organizational role, professional experience, organization size and development process. Table 4.1 shows the percentage and total number of responses to the question of "What is your primary role in your organization?". The majority of respondents were interaction designers or usability specialists (126), but we also collected a significant number (170) of responses from programmers and system analyst/designers.

What is your primary role in your organization?	**%**	**Total**
Usability Expert, Interaction Designer	**34.0%**	**126**
Systems Analyst/Designer	24.3%	90
Programmer	21.6%	80
CIO, Project Manager	10.5%	39
Other	9.5%	35
	100.0%	**370**
How many years of professional experience do you have in that role?		
< 2 years	20.3%	75
3-6 years	**40.8%**	**151**
7-10 years	22.7%	84
> 10 years	16.2%	60
	100.0%	**370**
How many information systems employees does your organization have?		
Less than 5	15.2%	55

5 to 14	16.3%	59
15 to 49	15.1%	55
50 to 100	11.0%	40
More than 100	**42.2%**	**154**
	100.0%	**363**
How would you classify your organization software development process?		
Just do it	14.0%	61
Waterfall	16.7%	54
Spiral	6.3%	38
Evolutionary development (Prototyping)	**25.3%**	**84**
Exploratory development	3.6%	17
Formal methods specifications	14.5%	43
Composition through reusable components	6.3%	18
Other	13.1%	48
	100.0%	**363**

Table 4.1 Respondent's demographic characteristics.

Regarding professional experience, 40.8% of our respondents declared 3-6 years of experience in their current role. As for their organization size, most respondents (42.2%) worked at large organizations, with more than 100 information system employees.

In terms of process, we measured responses for seven different development processes, defined as follows:

- *Just do it*, ad-hoc development, following no clear or well-defined process;
- *Waterfall*, which assumes a linear progression of development activities, where one begins only when the its predecessor is complete;
- *Spiral*, which combines development activities with risk management to minimize and control risk;
- *Evolutionary development (Prototyping)*: prototypes are developed and form the basis for some or all of the delivered software;
- *Exploratory development*: also based on prototyping, but where throw-away prototypes are made to explore the feasibility or desirability of possible solutions;
- *Formal methods specifications*: based on mathematical formalisms, specifications are produced and can be evaluated using proofs and (sometimes) automated techniques;
- *Composition through reusable components*: development is done bottom-up, building the remaining system around the available reusable components.

It was clear that evolutionary prototyping is most common development process (25.3%). This was not surprising, given the popularity of this development process.

4.6.3 Tools and tool usage patterns

We were interested in assessing which tools are currently being used to help interaction designers and software developers. We asked respondents to answer the question *"Which tool(s) do you currently use to perform user interface design? (Check all that apply to any user interface-related activity)."*, by checking all tools that they significantly used.

The tools studied were the following:

- *Paper and Pencil*;
- *Synchronous collaborative tools*: tools that allow people to work at the same time but possibly different places, like video-conferencing or instant messaging;
- *Formal model-based tools*: based on task models and focused on automatic code generation;
- *HTML-editing tools*: any tool that can be used to edit HTML documents, like MS Frontpage or Adobe's Dreamweaver;
- *Multimedia authoring tools*, e.g. Adobe Flash, Director;
- *Whiteboards*;
- *Analysis and modeling tools*: usually based on a modeling language such as the UML. Examples of this kind of tool are Rational Rose, MS Visio or Enterprise Architect;
- *Asynchronous collaborative tools*: tools that allow people to work at different times, like CVS systems or e-mail clients;
- *Electronic sketching tools*: tools that interpret sketches into a digital format, like SILK or DENIM;
- *Visual interface builders*: the common type of tool for designing UI's in a WYSISWYG manner, like MS Visual Studio or Apple's Interface Builder;
- *Post-it notes*.

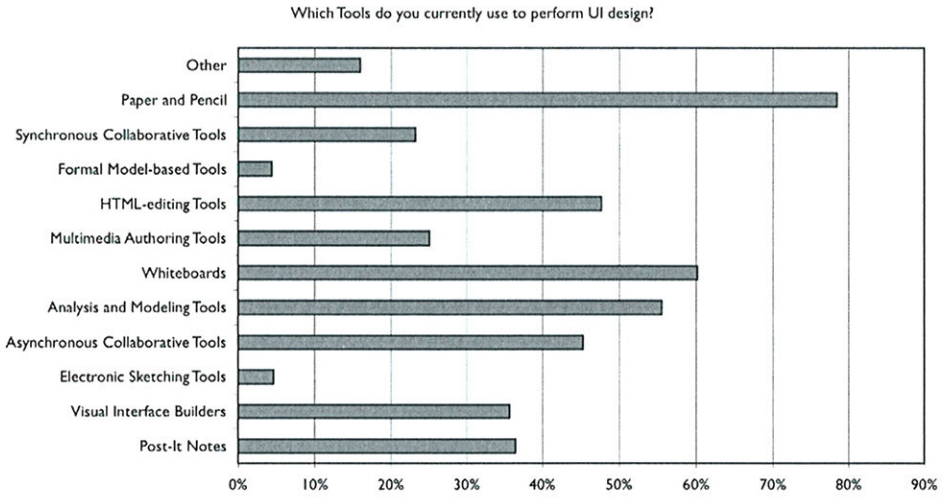


Figure 4.22 Kinds of tools used by practitioners.

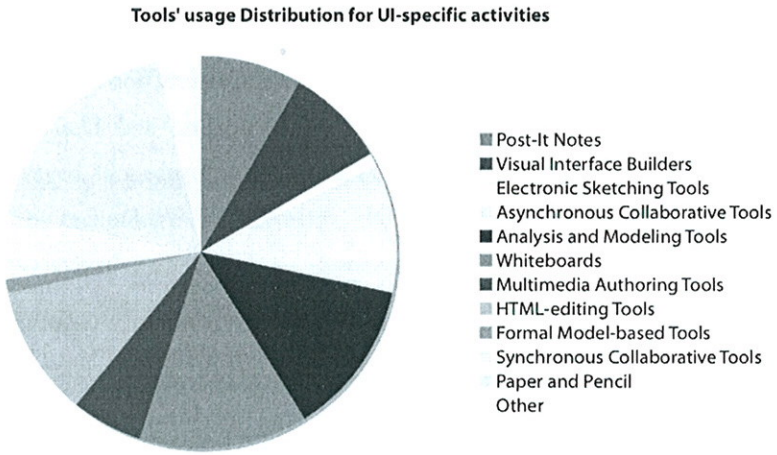


Figure 4.23 Distribution of the Tools Used.

Figure 4.22 summarizes our results, as well as Figure 4.23, which shows the same results in percentage, in the form of a pie chart.

Perhaps not surprisingly, paper and pencil was the most referenced tool, followed by whiteboards, analysis and modeling tools, HTML editors, asynchronous collaborative tools, "post-it" notes and visual interface builders. Formal model-based tools, although popular

in general software development, were not regarded as being critical to UI-related activities. Results also suggest that, regarding synchronous workstyles, designers typically work at the same time and place (by using whiteboards) rather than at the same time but different places (by using e.g. messenger, video-conferencing). Regarding asynchronous workstyles, almost half of the respondents use tools such as CVS and e-mail to perform UI-related activities.

It is interesting to note that Visual Interface Builders was ranked in 7 out of 12 classes of tools. This is surprising, since the survey specifically asked for the most used tools for UI-related activities. This suggests that despite the success of current Visual Interface Builders, there is a trend towards using widespread and particularly informal tools for interaction design. We believe this reflects the nature of interaction design, which is a highly creative, inherently interdisciplinary and communication-intensive activity. We argue that meshing formal, digital tools with informal, flexible, collaboration tools is needed in order to support the work of designers.

We also studied the relationship between the development process and the type of tools used. Table 4.2 and Figure 4.24 depict a subset of the results, for the following tools: Interface Builders, Asynchronous Tools, Analysis, Modeling and Design (AMD) Tools, Whiteboards, Synchronous Tools and Paper & Pencil.

	Interface Builders	*Asynchronous Tools*	*Whiteboards*	*Synchronous Tools*	*Paper & Pencil*
Just do it	39%	36%	54%	16%	59%
Spiral	34%	29%	61%	11%	82%
Prototyping	37%	45%	55%	23%	80%
Formal Methods	30%	56%	47%	26%	81%
Waterfall	46%	39%	69%	28%	80%

Table 4.2 Kinds of tools used by practitioners according to the most popular development processes.

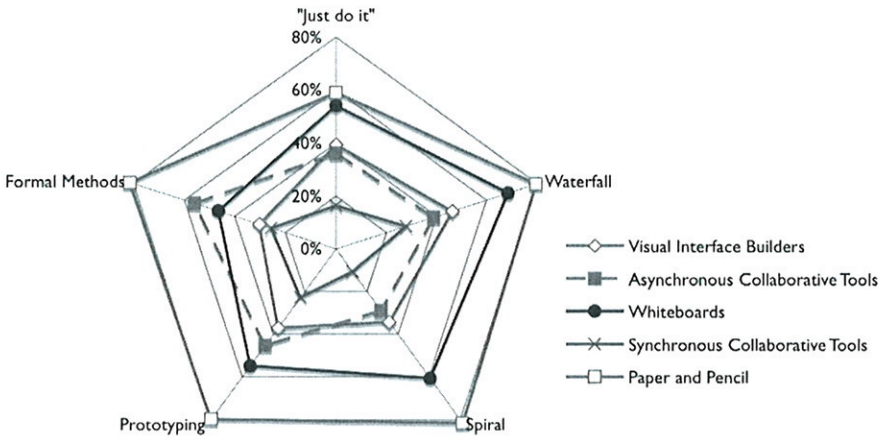


Figure 4.24 Tools usage plotted for the most popular development processes.

We found that together with Whiteboards, Paper and Pencil are the two most commonly used tools for all process styles except formal methods where asynchronous tools prevail over whiteboards. The less tool intensive development process is clearly the "Just do it" approach. We believe this reflects the urge towards coding activities that rely mainly on tools not surveyed, like editors, compilers and debuggers.

Despite all the technology enhancements, the low-tech Paper and Pencil and Whiteboards are, without any doubt, the most commonly used tools. The only exception is on "Just do it", perhaps reflecting the increased dependency on actual code for communication.

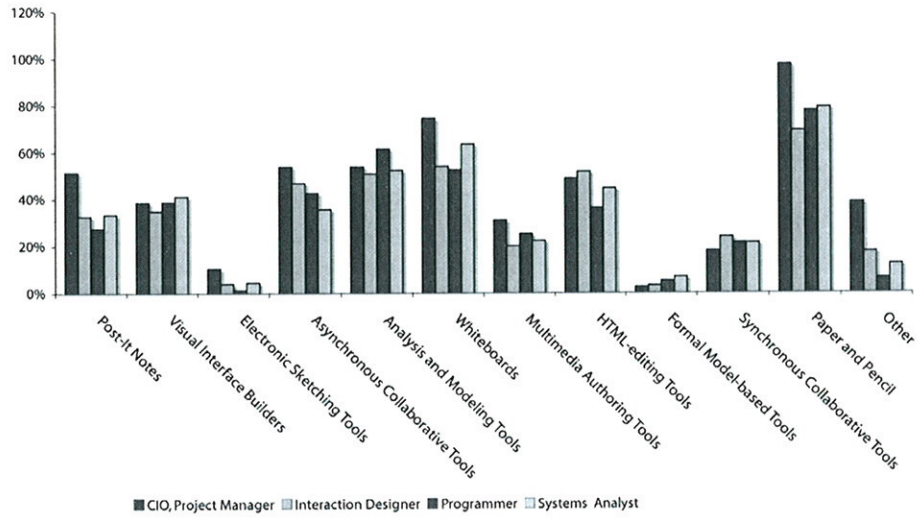


Figure 4.25 Usage of the tools divided according to the respondents' organizational role.

Figure 4.25 plots the tools' usage according to each of the four organizational roles. The largest difference was found among CIO's/Project Managers, which reported highest use of informal tools (Whiteboards and Paper and Pencil) than other respondents.

4.6.4 *Workstyle Transitions' Frequency and Cost*

The UI-related activities of interaction and software designers clearly suffer from a lot of interruptions that cause cognitive breakdowns. Designers are constantly being interrupted by colleagues, by e-mails, changing from individual to collaborative workstyles many times a day, or changing from low-tech card sorting to high-tech modeling with digital tools, something that also happens in general software development (Seffah and Kline, 2002; Wu et al., 2003).

We wanted to know what professionals thought about this, so we confronted respondents with several, concrete, scenarios of *transitions* in a workstyle, e.g. "Moving from high-level descriptions of the user interface (sitemaps, navigation maps, etc.) to detailed screens (with concrete widgets, buttons, etc.)". We asked them to rate the *frequency* and *cost* of those transitions. By *frequency*, we meant "how many times [the respondent] engages and transitions in those workstyles", and by *cost* we meant "how difficult [the respondent] finds

to perform that transition" (subsection 4.3.9 introduces these definitions). Participants answered this qualitative measurement by selecting a value from a 7-point Likert scale. Figure 4.26 plots the average over the total number of responses.

From Figure 4.26, we can verify that the second most frequent transition is "Moving from high-level descriptions of the user interface (sitemaps, navigation maps, etc.) to detailed screens (with concrete widgets, buttons, etc.)". Interestingly enough, this is also one of the transitions with lower cost (only exceeding "Moving from a Whiteboard to a CASE tool"). The second most costly transition was "Moving from non-functional to fully-functional prototypes".

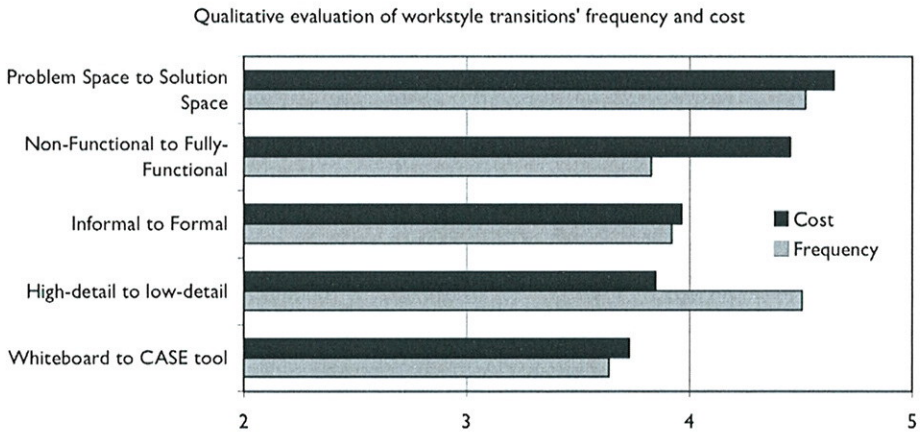


Figure 4.26 Some transitions in workstyles and their frequencies and cost.

The transition ranked the highest in terms of both frequency and cost was "Moving from business rules, use cases and problem space concepts into final solution design, and back". "Moving from non-functional prototypes into fully-functional prototypes" was also regarded as a costly transition, although less frequent, suggesting that functionality is added to prototypes in clearly defined stages of evolution, when a designer already has committed to some design decisions.

From the data collected, we were also able to cross-tab the results according to the professional role of the respondent (CIO/Project Manager, Programmer, System Analyst and

Interaction Designer). Figure 4.27 presents the results also in the form of a radar chart in order to visually perceive some interesting patterns.

We can see, for example, that the most significant difference between the roles' responses was related to the frequency of the "Low-detail to High-detail" workstyle transition. This frequency is perceived by interaction designers as being much higher than by any other role.

Another pattern that appears from the chart is related to programmers. Apparently, these are the respondents who gave the lowest classifications to the workstyle transitions presented, both in terms of frequency and in terms of cost. This does not mean, however, that workstyle transitions are less important in the work practices of programmers. It only means that for the transitions we presented, programmers were the ones who less viewed those transitions as frequent and/or costly.

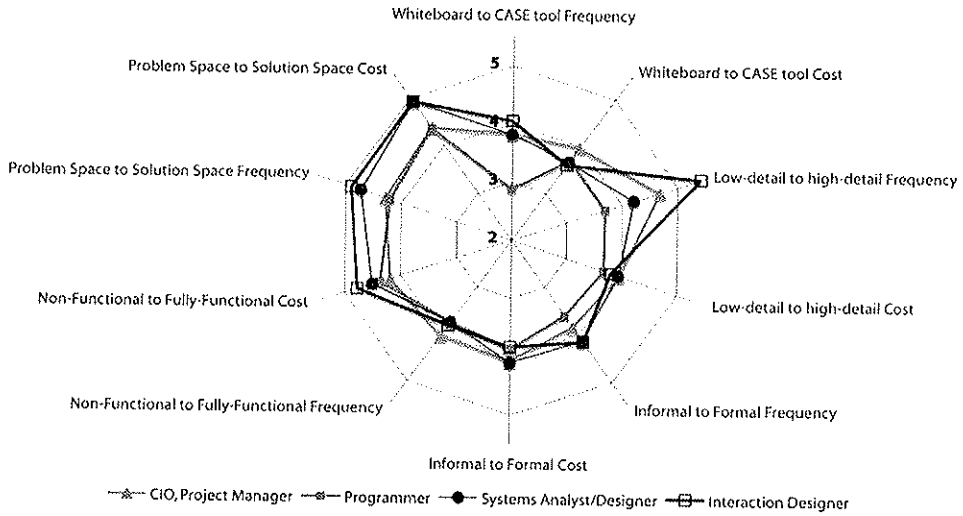


Figure 4.27 Frequency and Cost of Transitions plotted for the different roles: CIO, Programmer, Systems Analyst/Designer and Interaction Designer.

Finally, and following the recommendations of (Dillman, 1999), we performed follow-up interviews. In one of those interviews (results were very similar in the other interviews), a designer stated that his answers "would have been different if he was being questioned about issues such as debugging or code maintenance.". His answer was very similar to other respondents interviewed, who also stated "the importance of building better tools for

UI-related activities" and "how transitions were more difficult in UI-related issues". We believe this helps to reinforce our claim about the importance of supporting workstyle transitions in UI activities.

4.7 Applying Workstyle Modeling to other Domains

In order to assess the similarity of our work with related approaches, we participated at the Interact'05 Workshop on Human Work Interaction Design (Clemmensen et al., 2005) where we discussed whether similar workstyle models could be designed and applied to domains other than User-Centered Design tools.

Wu and Graham (2004) have already successfully conceived and applied a workstyle model for collaborative software design. Work-centered approaches and frameworks have been described in, e.g., (Fidel and Pejtersen, 2004; Pejtersen, 1992, Vicente, 1999). In this chapter and in (Campos and Nunes, 2005), we have described a model that captures the workstyles of a team of interaction designers. We then questioned whether workstyle modeling could be transformed into a general method of human-work interaction design. Figure 4.28 shows for the first time a lightweight methodology that can be followed in order to incorporate workstyle modeling into the design process.

As an example, consider the design of a system for selling tickets at an Arts Center[3]. The designer first identifies the important dimensions in the users' workstyles. In this case, a ticket-selling agent can work facing a large queue of customers or a small number of them. She can sell tickets over the phone in the morning or face-to-face at the ticket window during the afternoon. She may be facing a long queue of clients and then suddenly attending just one or two every hour. She frequently changes from selling tickets to delivering pre-paid tickets.

One could plot a workstyle model with three dimensions: Location (over the phone or at the ticket window), Queue Size (small or large), and Function (selling or delivering), like Figure 4.29 shows. Now, we can identify the most important transitions. The importance of a transition is related to its frequency. If the selling agent switches frequently between selling and delivering, the system should provide these two functions in the same interaction context.

---

[3] This example was kindly provided by Larry Constantine, who used it throughout a UsageCD course at the University of Madeira.

Thinking about the transitions between values in those dimensions can give rise to new design ideas as well as a means to validate design decisions: changing from selling tickets at the ticket window facing a large crowd to providing information over the phone can be supported efficiently in the system? This way of thinking can also aid in achieving a tighter fit between everyday work practices and the new system.

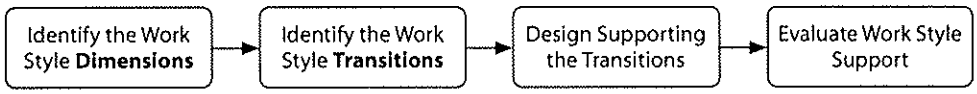


Figure 4.28 Work Style Modeling as a Design and Evaluation Aid.

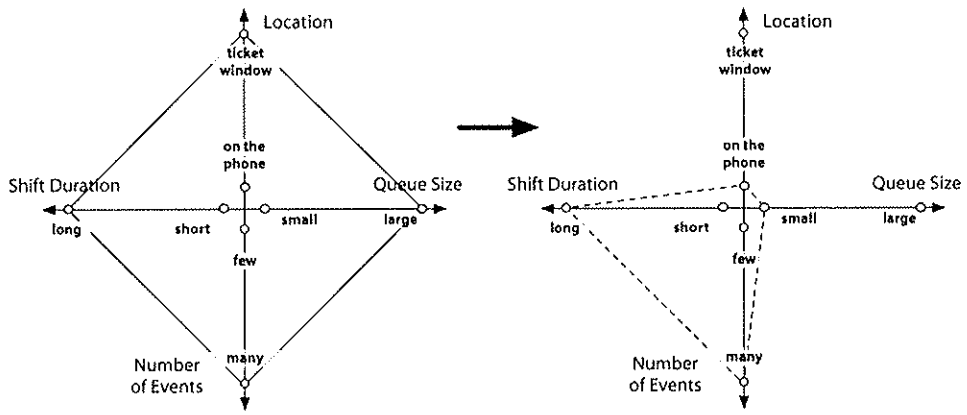


Figure 4.29 A model for workstyles in the Arts Center work domain: a transition example.

It is our belief that there is a need for more elaborate models describing users in work, because current methods don't adequately support flows or transitions between contexts of work. We think such models could be devised not only for requirements and analysis, but also as a discussion tool to be used during the whole process.

Describing and modeling the styles of work users were engaged with has helped us design more usable UCD tools, such as CanonSketch (Campos and Nunes, 2004; Campos and Nunes, 2005b; Constantine and Campos, 2005). Usability studies have shown better results and we believe workstyle modeling can be very useful both as an informal discussion tool and as a Human-Work Centered Design approach.

4.8 Conclusions

In this chapter, both the motivation and the foundations of this thesis were presented. The workstyle model is essentially a new way of describing and capturing information about workstyle transitions. We briefly outlined how workstyle modeling (in general) could be abstracted, generalized and tailored to be an efficient, practical modeling guide in other design domains.

To gather information and reinforce our claim about the importance of workstyle transitions, and also to study the tools that are actually used, in professional, real world settings, we designed and performed a survey. The answers to our survey, which were obtained anonymously from a significant number (370) of professionals engaged in UI activities clearly showed that the workstyle transitions are perceived as being significantly difficult and frequent throughout their daily tasks. By significant, we mean above the 3.5 neutral value in the 1-7 Likert scale presented to respondents, and by difficult, we mean in terms of average, perceived effort required to accomplish the transition. The study also shed light on the tools that are currently being most used by practitioners regarding UI or UCD practices. Namely, the study revealed that low tech tools such as paper and pencil, whiteboards and post-it notes are still significantly used, regardless of the respondents' organizational roles. In fact, we discovered that together with Whiteboards, Paper and Pencil are the two most commonly used tools for all development processes except formal methods where asynchronous tools prevail over whiteboards.

Regarding synchronous workstyles, the results of our study suggest that practitioners perform UI-related activities at the same time and place rather than at same time but different places. Whiteboards were said to be used by 60% of respondents, whereas tools for working at the same time but different places (like messenger, video, etc.) were referenced just by 23%.

Another clear result is related with current visual interface builders. These were referenced as being used only by 36% of respondents. We believe this shows that despite the success achieved by this class of tools, more research has to be performed into how visual interface builders can better support informal human collaboration. We gave some simple examples of how to work towards this goal by presenting the sketch recognition engine in

CanonSketch as well as the layer for informally annotating and sketching the final UI. With the probable dissemination of technology such as smartboards, as well as new developments in paper-like display technologies, we believe future tools will become more usable and will eventually replace the current massive use of paper, pencil and whiteboards (80% and 60% respectively in our survey).

Finally, another interesting conclusion is that the transition regarded by our 370 respondents as being both most costly and most frequent was "Moving from business rules, use cases and problem space concepts into final solution design, and back". Going from problem space to the solution space is clearly the hardest of all workstyle transitions. To support this, we believe that more integrated modeling tools have to be designed. We will give in CHAPTER 5, as an example, the TaskSketch tool, which tries to provide traceability between use cases, the system's conceptual architecture and the UI's initial layout elements.

We believe that this kind of qualitative research, performed empirically by asking professionals to qualitatively classify cost and frequency of particular development scenarios is very useful for understanding current work practices. And understanding the everyday tasks of software engineers and interaction designers is a solid foundation for building a better world of tools, and ultimately removing the "hate" from the love/hate relationship between practitioners and their tools.

There is ample room for innovation regarding tool support for UCD processes. Current tools don't fulfill (at least totally) the UI activities of their users: the developers and interaction designers. Support for collaboration and informal communication is even more critical in processes such as UCD. More importantly, supporting transitions in workstyle dimensions can lead to better user-centered tools for user-centered design, given the iterative, evolutionary nature of the process.

Our framework is the first workstyle model tailored to UCD. Current models are too specific and are not expressive enough to be applied more generally to UCD. In this chapter, we expanded them and unified their most significant dimensions in order to achieve a more useful and usable model that could convey more information in a better way. We provided a set of guideline questions that can be used to plot values in the model space. However, our model should be regarded more as an informal discussion tool, rather than a

formal method for analyzing workstyles. Contrary to other models, it allows estimating the effort and stage of development of a UCD process by checking the size of the region or line. Its dimensions were specifically designed to allow an easy and intuitive plotting of styles of work. Our model can also be effectively used to (a) choose adequate tool support for a given phase of a project and (b) drive the development of new UCD tools.

5 The Design of Design Tools

Designing for Workstyle Transitions

"If you can dream it, you can do it."

Walt Disney

It was Larry Constantine who excelled at describing the reasons why designing new tools for integrating software engineering with interaction design are needed: "software engineering models and methods, as well as the tools supporting them, have largely ignored UI design. Despite its ambitious moniker, the Unified Modeling Language (UML), a pastiche of discrete models of diverse heritage, also fails in this arena. The UML is particularly weak when it comes to expressing visual and interaction design, appropriating a Procrustean patchwork of models originally intended for other purposes" (Constantine and Campos, 2005).

The UML is also weak in a critical aspect of use cases, one of the core models of object-oriented software engineering (Jacobson et al., 1992). Widely employed for requirements modeling in software engineering, use cases are also commonly used in user-centered and usage-centered design for modeling user tasks.

The UML, as well as the software tools that support it, recognize use cases as modeling objects. However, their internal structure and content is not recognized. And unfortunately, it is these details of user interaction with software that are important drivers for effective UI design (Constantine and Lockwood, 1999; Constantine and Campos, 2005).

CanonSketch (Campos and Nunes, 2004a, 2004b, 2005a) and TaskSketch (Campos, 2005b, Constantine and Campos, 2005), the two main tools developed during this thesis, are presented, explained and illustrated in this Chapter. They position themselves as Inter-

action Design tools aimed at software engineers. It is a generally accepted notion that software engineers need better interaction design tools, tools that will help them create higher quality user interfaces. Besides the fact that turbulent, small software development companies don't have the budget for having dedicated interaction designers, it is also a fact of life that software engineers – who are not experts at interaction design – *will* create UIs.

What differentiates our design strategy from others – besides this line of argumentation focused on a software engineering perspective – is the fact that our approach is specifically aimed at *designing in order to support workstyle transitions*, in particular to support some of the most frequent and difficult transitions as reported by the professionals that answered our survey (described in the previous Chapter).

In this Chapter, we describe the most important aspects of CanonSketch and TaskSketch, in particular the workstyle transitions that are supported by several characteristics of the tools. We illustrate the discussion with relevant examples and case studies. The following paragraphs provide a roadmap for this Chapter.

In SECTION 5.1, we introduce and overview the tools and how they were developed.

SECTION 5.2 is entirely devoted to describing and illustrating the CanonSketch tool. The original requirements and the original idea for the tool are first presented; afterwards, the initial version is described. A linkage between Canonical Abstract Prototypes and the Wisdom (Nunes, 2001; Nunes and Cunha, 2000) had to be designed and is described in this section, followed by some examples. One of the most novel aspects brought by CanonSketch, the capability to express UI design patterns at several levels of detail, perspective and functionality, is also described and thoroughly illustrated with examples taken from several, different UI patterns collections. The foundation for the final version of CanonSketch is then outlined in the subsection devoted to the generation issues of MXML, a new XML-based language for defining interactive systems. We overview the final version of CanonSketch that was designed using the Design Research approach outlined in the Introduction of this thesis, and we demonstrate how it supports workstyle transitions. But since the mere description and illustration of complete examples supporting transitions does not provide the feel of the fluid experience of the tool, a complete application design and implementation example is also provided and discussed.

SECTION 5.3 is in turn dedicated to the "twin" application of CanonSketch, called TaskSketch. Among the particular issues focused, it is described how supporting perspective and traceability transitions is achieved using TaskSketch, how formality transitions are also possible and how collaboration can be fostered through the collaborative features of TaskSketch, namely the Brainstorming collaborative environment and the blog integration mechanisms.

SECTION 5.4 concludes the chapter with some discussion items that arise from the design process of these design tools.

5.1 Introduction

CanonSketch and TaskSketch are experimental tools that address the critical shortcomings of UML and UML-based tools and techniques, by aiming at providing support for a new UML-based version of Usage-centered Design. This new version is a mixture of the UML-based Wisdom notation (Nunes, 2001) with the classical UsageCD notations described in (Constantine and Lockwood, 1999).

These proof-of-concept tools were developed throughout the course of this thesis in Objective-C for Mac OS X and using object-oriented software engineering techniques, such as the Model-View-Controller pattern. Both tools use industry-standard object modeling notation (UML) and compatible extensions and integrate with standard tool suites through XMI export.

The initial version of CanonSketch was able to generate HTML as a proof of concept on the expressive power of canonical abstract prototypes. The later version introduced a smooth way of transitioning from abstract to concrete, non-functional to fully-functional prototypes, by adding support for MXML (Adobe, 2006) concrete UI elements that can be generated and manipulated, as we will describe in the next section. We used the Workstyle model both as a way of focusing the design and development of the tool in terms of smoothing transitions and as a way of evaluating, reflecting and justifying design decisions.

TaskSketch (Campos, 2005b; Campos, 2005a; Constantine and Campos, 2005) is an interactive requirements elicitation and modeling tool focused on linking and tracing essential use cases (EUC). An EUC, as described in §2.1.4 of this dissertation, is a use case representing a single, discrete end-user intention expressed in essential form, that is, as an abstract, generalized, simplified, and technology- and implementation-independent narrative (Constantine, 1995). TaskSketch supports collaborative modeling by multiple stakeholders, including clients, marketing staff, and software engineers. It is unique in facilitating the development and exploration of the conceptual architecture based on use case narratives developed in essential form. It enables tracing system requirements, in terms of user intentions and system responsibilities, to the conceptual architecture of the system,

making it easy to extract the software architecture from task flows and to prioritize development of classes.

5.2 The CanonSketch Tool

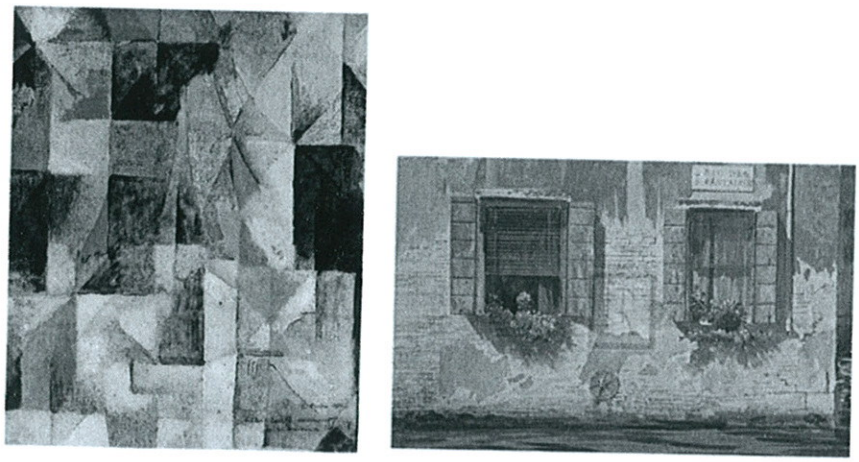


Figure 5.1 Two different ways of representing windows at two different levels of abstraction: on the left, R. Delaunay's very abstract "Fenêtres". On the right, the more concrete painting by M. Flemingham, "Venetian Windows".

To illustrate how our workstyle model can be used to envision innovative interaction and software design tools, we present in this section the CanonSketch prototype tool that supports interaction design at multiple levels of abstraction (or detail).

Since the advent of OO&HCI methods (van Harmelen, 2001) several approaches have been proposed to support interaction design using OO notations like the UML. In the WISDOM method (Nunes and Cunha, 2001), it is proposed to use stereotyped class diagrams to depict the presentation aspects of interactive systems (through the concept of interaction space). Although the WISDOM notation (Nunes and Cunha, 2001) could be supported in any UML modeling tool, experience showed that the tools were not particularly effective supporting UI design. As a result, developers adopting the WISDOM method usually shifted from traditional AMD tools to more informal tools seeking the design freedom associated with interaction design.

The original idea for the CanonSketch tool (Campos and Nunes, 2004a, 2004b; 2005b, 2006b) was then to support the WISDOM notation at the presentation level and also to combine UML class stereotypes with the Canonical Abstract Prototype (CAP) notation. Canonical Abstract Prototypes were developed by (Constantine and Lockwood, 1999), af-

ter a growing awareness among designers regarding the conceptual gap between task models and realistic prototypes. They provide a common vocabulary for expressing visual and interaction designs without concern for details of behavior and appearance. Moreover, they fill an important gap between existing higher-level techniques, such as the WISDOM UML-based interaction spaces and lower-level techniques, such as concrete prototypes. For this reason, and also because CAPs are implementation-independent, we chose them as the middleware notation to support interaction design in CanonSketch. In the following section, the different CanonSketch aspects regarding workstyle support are briefly described.

5.2.1 *CanonSketch's initial version*

Following our Design Research approach (March and Smith, 1995), we will describe the first artifact created as "a situated implementation", after which we extracted knowledge as operational principles. It is therefore important to describe both the first and the second version of CanonSketch, since one can obtain detail about what was changed and how the tool was actually improved.

In a paper presented at a recent Workshop on MB-UID (Trætteberg et al., 2004), we argued that in order to achieve a stronger market acceptance of modeling tools, a new generation of user-centric tools would have to emerge.

The existing tools are focused on the formalisms required to automatically generate the concrete user-interfaces. This legacy of formalism-centric approaches prevents the current tools from adequately supporting the thought and design tasks that developers have to accomplish in order to create usable and effective user-interfaces. Model based approaches concentrate on high-level specifications of the user-interface, thus designers loose control over the lower level details. These problems with MB-UI tools are described in (Myers, 2000). In particular, those tools suffered from trying to solve the "whole problem" and thus providing a "high threshold/low ceiling" result. As we have already described, 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. Thus, those tools don't concentrate on a specific part of the UI design process and are difficult to learn, while not providing significant results.

In order to overcome these limitations, designers directly use a user interface builder that provides them with adequate and flexible support for designing the user interface. Designers that recognize the value of modeling at high levels of abstraction are forced to use different tools and notations to capture the user-interface specifics at different levels of abstraction – what could be considered as using many low-threshold/low ceiling tools.

Some of the requirements for such tools were also discussed in a recent workshop about usability of model-based tools (Trætteberg et al., 2004). Among other issues, the participants at the workshop highlighted the following requirements as paramount to promote usability in tools: traceability (switching back and forth between models, knowing which parts can be affected by changes), support for partial designs, knowledge management (for instance, a class that is selected or modified often is probably more important than classes not often changed) and smooth progression from abstract to concrete models.

The goal of the initial version of CanonSketch (Campos and Nunes, 2004a, 2004b) was to try to leverage the users' previous experience with popular Interface Builder (IB) tools in order to achieve better adoption levels. Our aim was to build a developer-centric modeling tool that applied successful concepts from the most usable and accepted software tools. Instead of defining a complex semantic model and formalisms to automatically generate the user interface (UI), we started by using a simple sketch application and extending it to accommodate the required concepts and tools. Thus, the initial version of the tool supported the creation and editing of Canonical Abstract Prototypes (Constantine, 2003) and WISDOM Presentation Models (Nunes and Cunha, 2000). It was capable of automatically generating HTML interfaces from the Canonical specification. In this initial phase, we were focusing on specifying GUI's for Web-based applications, although conceptually the tool was not restricted to this type of interface, since the languages are platform- and implementation-independent. However, this phase did allow us to test the main concepts of the tool/language by focusing on a well-known interface type.

Figure 5.2 shows one of the screenshots of the tool: the Wisdom UML view, where the designer is creating a Wisdom presentation model as if she was sketching in a simple drawing application. Figure 5.3 shows a screenshot of the early Canonical view: we can see that there are several palettes of tools available (e.g. for controlling font-size, coloring and grid layout) and an inspector, as well as an optional ruler.

In our path to building a usable modeling tool for UI design, we began with a different approach from the conventional way these tools are envisioned: instead of focusing on the formalisms and semantics, we began with a simple drawing application and built a modeling tool that relies on interaction idioms more closely related to Office applications, as we discuss in the following sections. Our remit here is that we intend to focus on achieving a modeling tool that is as easy to use as a drawing application.

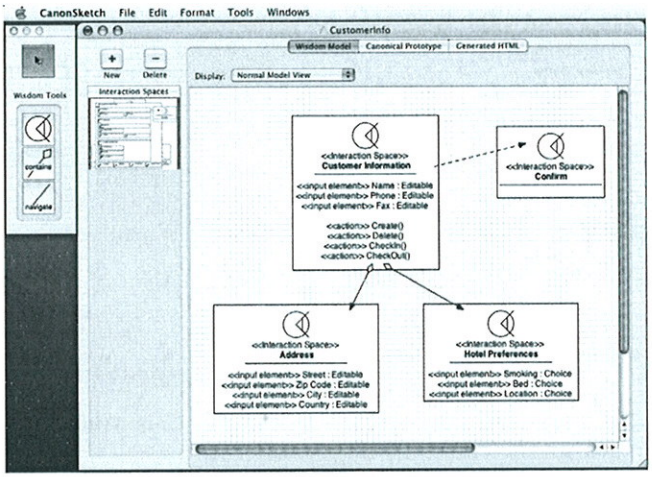


Figure 5.2 One of the earliest screenshots of the CanonSketch tool. Shown here is the Wisdom UML view.

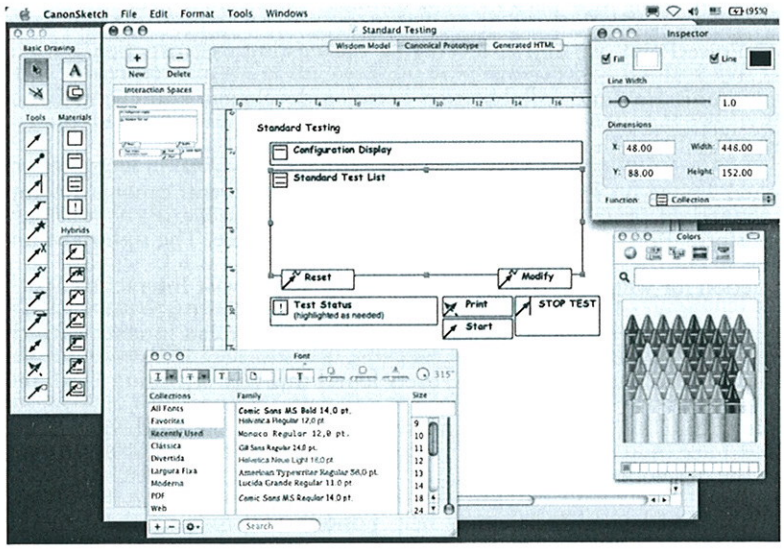


Figure 5.3 The intermediate version of CanonSketch showed the miniature CAP's a la MS PowerPoint, as well as the CAP, UML and HTML views.

User-Centered Features. UI tools represent an important segment of the tool market, accounting for 100 million US Dollars per-year (Myers et al., 2000). However, there has been a gross decline on the modeling tools market revenue, according to reliable sources such as the International Data Corporation (Robbins, 1999). The lack of usability present in modeling tools is believed to be responsible for this weak adoption (Trætteberg et al., 2004).

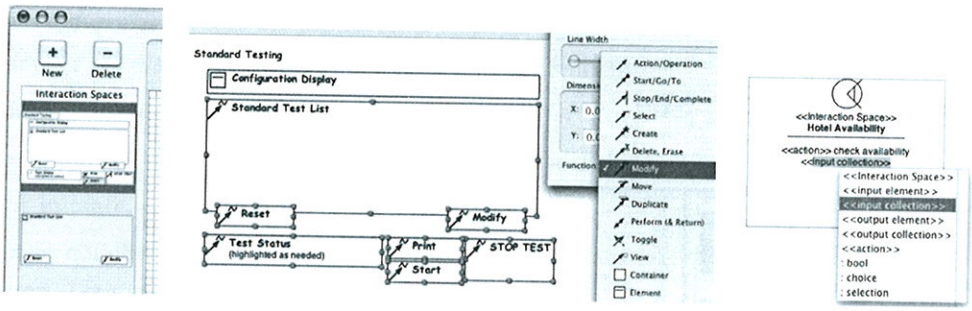


Figure 5.4 Features of the initial version of CanonSketch.

A more developer-centered approach was followed in CanonSketch: Figure 5.4 shows some of the aspects we took into account. Canonical Abstract Prototypes are organized in terms of sequences of interaction spaces that appear as thumbnails of their corresponding specifications. By using this pattern, very common on business presentation applications such as MS PowerPoint, we aim at leveraging the existing user experience while also promoting communication and collaboration between developers and clients (who are often businessmen familiar with this pattern).

The center image on Figure 5.4 shows a selection of several canonical components to apply a transformation of their interactive function all at once. The rightmost image shows code completion for when the designer is specifying a WISDOM Interaction Space (which is a UML class stereotype representing "space" where the user can interact with the application). During informal usability studies, users found this way of editing UML models to be more usable than filling in complex forms that only update the UML view after validating everything the developer introduced.

Finally, the grid layout option may help position and resizing the components more rapidly, and the tool palettes follow the pattern of the successful Interface Builders.

Tabbed-view navigation is important in order to achieve model linkage at the various stages of the process.

A proof of feasibility: Generation of HTML Forms. There was a third view in the first version of CanonSketch where a concrete prototype, in HTML form, was automatically generated, thus illustrating one possible concrete implementation. The concrete prototype is fully navigational, since it is rendered using an embedded, fully functional web browser, as we can see in Figure 5.5.

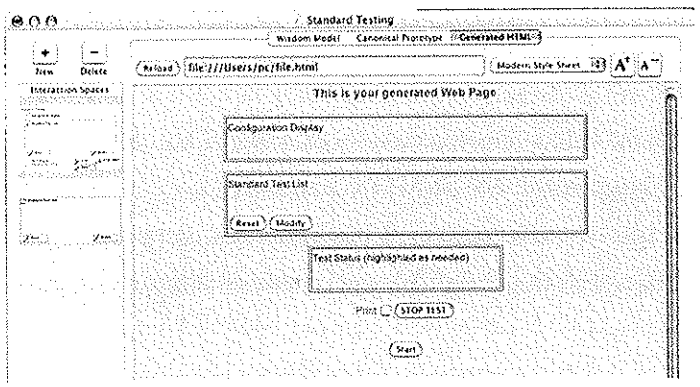


Figure 5.5 Simple HTML automatically generated from the specification in Figure 5.3.

In order to verify the richness of the notation developed by Constantine and colleagues, and also to support automatic generation techniques, still without a semantic model defined, we built a proof of feasibility that can be exemplified in Figure 5.5. The HTML form shown was automatically generated from the canonical specification illustrated in Figure 5.3.

The HTML clickable prototype is useful for rapidly testing the navigational structure of the specified interface. The tool can also generate a PDF printable version of the Canonical/Wisdom models, which can act as a means to document the development process and commit to design decisions made with the client.

In the absence of a semantic model incorporated into our tool, this proof of concept already showed the potential of the notation, and achieved our first, initial goal of checking the richness of the abstract prototype notation. This is also part of our approach based on

starting from a usable, simple tool and successfully add semantic mechanisms in an incremental way, rather than building a complex, formalism-centric tool.

5.2.2 *The WISDOM UML Profile for UI Design*

The Wisdom notation (Nunes and Cunha, 2001) is a set of UML-compatible notations (a UML Profile) supporting efficient and effective interactive systems modeling. In the CanonSketch project we refined the Wisdom notation to achieve a full semantic support for Canonical Abstract Prototypes (CAPs), as well as to maintain synchronized UML/CAPs views (by means of a common semantic model). The specification of such a common model allowed the tool to support the design process at different levels of Detail. It also complements the weaknesses of one notation with the strengths of the other.

To support the modeling of presentation aspects of the UI, the Wisdom method proposes the following extensions to the UML (Nunes, 2001):

- «Interaction Space», a class stereotype that represents the space within the UI where the user interacts with the all the tools and containers during the course of a task or set of interrelated tasks;
- «navigates», an association stereotype between two interaction space classes denoting a user moving from one interaction space to another;
- «contains», an association stereotype between two interaction space classes denoting that the source class (container) contains the target class (contained); The «contains» association can only be used between interaction space classes and is unidirectional.
- «input element», an attribute stereotype denoting information received from the user, i.e., information the user can operate on;
- «output element», an attribute stereotype denoting information displayed to the user, i.e., information the user can perceive but not manipulate;
- «action», an operation stereotype denoting something the user can do in the concrete UI that causes a significant change in the internal state of the system.

In CanonSketch, the designer is able to choose between the UML model view and the CAP view, and switch back and forth while maintaining coherence between the models. This adds support for the Traceability dimension in our workstyle model.

Although Canonical Abstract Prototypes lack a precise formalism and semantics required to provide tool support and automatic generation of UI, we found the notation expressive enough to generate concrete user interfaces from abstract prototypes. Our tool presents a proof of feasibility, since we generate HTML pages from sketches of Canonical Abstract Prototypes. We also found the notation very useful for expressing design patterns in an abstract, platform independent way.

5.2.3 Linking Wisdom UML to Canonical Abstract Prototypes

Experience using the WISDOM notation in different settings enabled to identify some problems with using stereotyped class diagrams to capture the presentation aspects of interactive systems. In particular, spatial information and other lower level aspects are important for some common UI patterns. As discussed in (Nunes, 2003) when applying the WISDOM approach to UI patterns, some problems derive from detailed presentation aspects, such as size, position, or use of color. Specifying a linkage between Canonical Abstract Prototypes and the WISDOM Presentation Model can help solve some of these problems, while also adding the necessary formalism to the Canonical notation.

In Figure 5.6, we show the specification of a mapping between the WISDOM Presentation Model and Canonical Abstract Prototypes. An interaction space in WISDOM was mapped as an interaction context in a Canonical Prototype, since both definitions are similar (see Constantine and Lockwood, 1999; Nunes and Cunha, 2000). Although not present in Figure 5.6, the WISDOM «navigates» association can be unidirectional or bi-directional; the latter usually meaning there is an implied return in the navigation. This has the same meaning Constantine defines when describing the UsageCD contexts' navigation map (Constantine, 2003): a map that "defines the interconnections between among the various interaction spaces of the user interface architecture". An «input element» attribute stereotype is mapped to a generic active material, unless typified. Input elements specify information the user can manipulate in order to achieve a task.

An «output element» attribute stereotype maps to an element and an «action» operation stereotype to an action/operation Canonical component. The «contains» association stereotype is mapped to a Canonical container. We can also see from Figure 5.6 that our

extension to the WISDOM presentation model notation fully supporting Canonical Abstract Prototypes consists in adding two more attribute stereotypes:

- «input collection», an attribute stereotype denoting a set of related information elements received from the user, i.e., a set of input elements; an «input collection» can be used to select from several values in a drop-down list, or choosing one element from a table to perform any given operation;
- «output collection», an attribute stereotype denoting a set of related information elements displayed to the user, i.e., a set of output elements. Typically, an «output collection» conveys information to the user about a set of elements of the same kind, for instance a search results list or the results display from a query to a database.

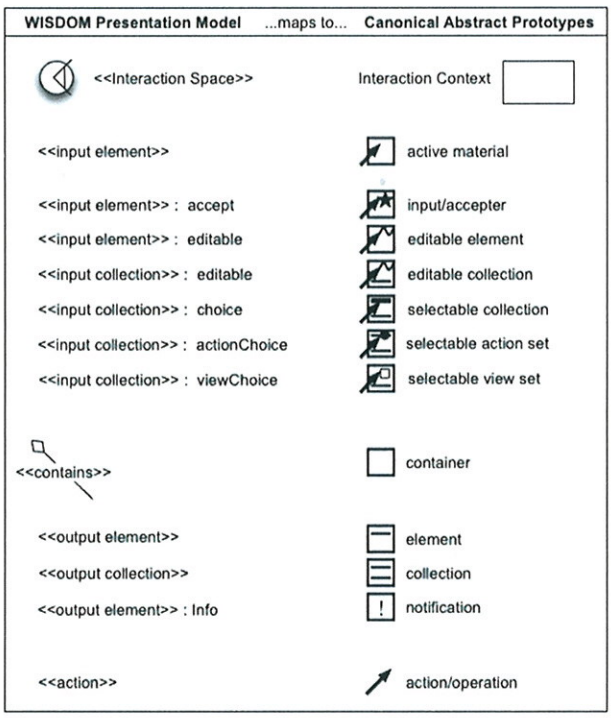


Figure 5.6 Extending the Wisdom profile to support Canonical Abstract Prototypes: this figure shows the correspondence between Wisdom UML stereotypes and Canonical components.

By typifying these attribute stereotypes, one can map a WISDOM presentation model to all Canonical components that belong to the classes of Materials or Hybrids. For instance, an input collection typified as choice can be mapped to a selectable collection. The de-

signer starts by specifying the general structure of the UI using a UML extension (the WISDOM notation). That specification is mapped to one or more Canonical interaction contexts, where the designer expands and details the model in terms of size, position and interactive functions. This mapping clearly shows the role of WISDOM interaction spaces realizing the interface architecture, and how it can be combined with the Canonical notation to help bridge the gap between abstract and concrete models of the user interface.

Figure 5.7 shows an example of a WISDOM Presentation Model for a Hotel Reservation System (described in and taken from (Nunes, 2001)). Figure 5.8 depicts a Canonical Abstract Prototype that corresponds to the area inside the dashed rectangle in Figure 5.7. This mapping clearly shows the role of WISDOM interaction spaces realizing the interface architecture, and how it can be combined with the Canonical notation to help bridge the gap between abstract and concrete models of the user interface.

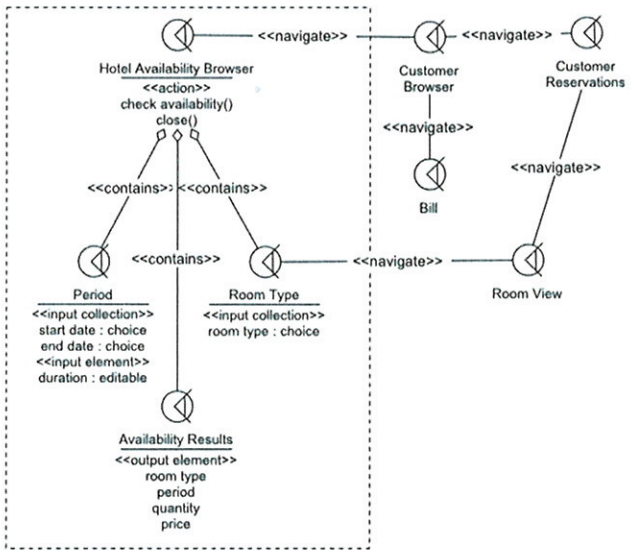


Figure 5.7 A Wisdom Presentation Model for a Hotel Reservation System (described in and taken from (Nunes, 2001)).

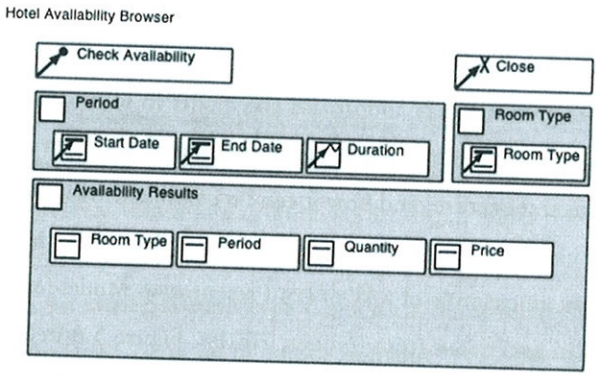


Figure 5.8 A Canonical Abstract Prototype for the same Hotel Reservation System as in the area inside the dashed rectangle in Figure 5.9.

5.2.4 *Using CanonSketch to Communicate UI Patterns*

The capability of identifying UI patterns and expressing the solution in an abstract way independent of any particular platform or implementation is becoming more and more important, with the increase in the number of information appliances. The WISDOM notation enables an abstract definition of UI patterns (Nunes, 2003), and also complies with the UML standard. However, some problems remain for patterns expressing more concrete presentation aspects, such as size or positioning.

Having a tool that provides a common semantic model linking Canonical components to Wisdom elements can help solve some of these problems. It also adds the required formalisms for generating concrete user interfaces from Canonical specifications. Our expectation was to incrementally build such a tool from this initial version of CanonSketch.

Since the Canonical Abstract Notation is a way to express visual design ideas that was devised to support decision-making at a higher level of abstraction than concrete prototypes, we tried to investigate the ability to express GUI design patterns using CanonSketch.

In this section, we present some examples of the WISDOM notation extension applied to some GUI patterns (taken from the Amsterdam collection (Welie and Trætteberg, 2000)) and also the Canonical representation for the same patterns. As Constantine points out, "the ability to express design patterns in terms of generalized abstract models has seen little use in UI patterns" (Constantine, 2003). We still lack some widely accepted notation to

represent commonly used solutions to some interaction tasks in an abstract way that can be applied to many design scenarios (Nunes, 2003).

Throughout this section, all the Figures illustrate a Final User Interface (FUI) linked to a Concrete User Interface (CUI) or Abstract User Interface (CUI), in the terms defined in § 3.6.1. The FUI is represented by a screenshot of a particular implementation of the pattern, and the AUI is represented by the Canonical and WISDOM representations.

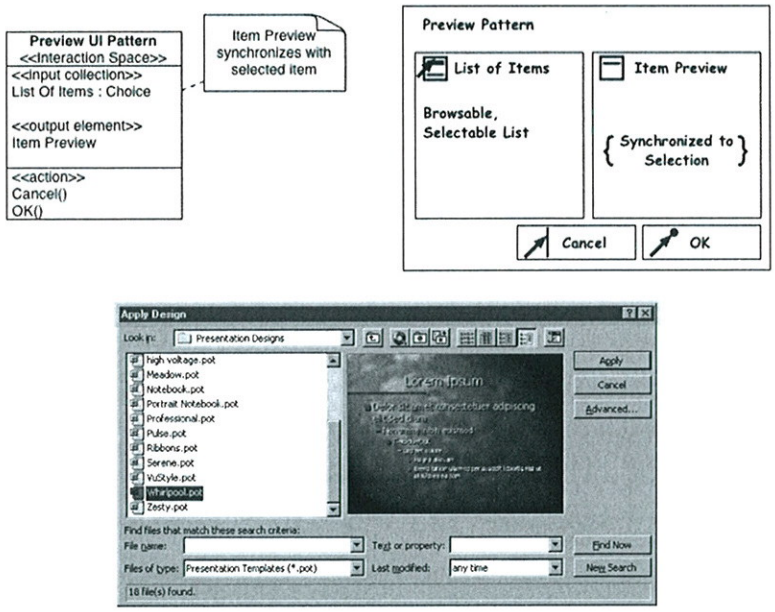


Figure 5.9 A Wisdom (top left) model, a Canonical prototype (top right), both applied to the Preview Pattern. A concrete example is shown at the bottom: a dialog from MS PowerPoint.

In Figure 5.9, we present the WISDOM and Canonical representations for the GUI Preview pattern (Welie and Trætteberg, 2000). We also present a concrete realization of this pattern (a dialog from MS PowerPoint). The problem this pattern tries to solve occurs when the user is looking for an item in a small set and tries to find the item by browsing the set. This pattern is particularly helpful when the items' content nature does not match its index (e.g. a set of images or audio files are indexed by a textual label). The solution is to provide the user with a preview of the currently selected item from the set being browsed (Welie and Trætteberg, 2000). As we can see, there is not much difference in this case. On the one hand, the WISDOM representation (on the top left), is much more com-

pact, because it is based on the UML. But the Canonical representation has the advantage of clearly stating that the browsable list of items is placed to the left of the item preview, which conforms with the western way of reading and therefore adjusts to the task being performed: the user first selects an item, and only then he focuses on the preview. It is also evident that the Canonical notation is much closer to the concrete representation of this pattern (at the bottom of Figure 5.9).

In the following pattern, the advantages of combining both WISDOM and Canonical representations are also evident. The grid layout pattern, also from the Amsterdam collection (Welie and Trætteberg, 2000), tries to solve the problem of quickly understanding information and take action depending on that information. The solution is based on arranging all objects in a grid using a minimal number of rows and columns, making the cells as large as possible (Welie and Trætteberg, 2000). The bottom of Figure 5.10 shows an example of a concrete GUI where this is achieved (a dialog box from Word 97). By using this pattern, screen clutter is minimal and the layout is more consistent. The top of Figure 5.10 shows the WISDOM representation at the left and the Canonical representation on the right.

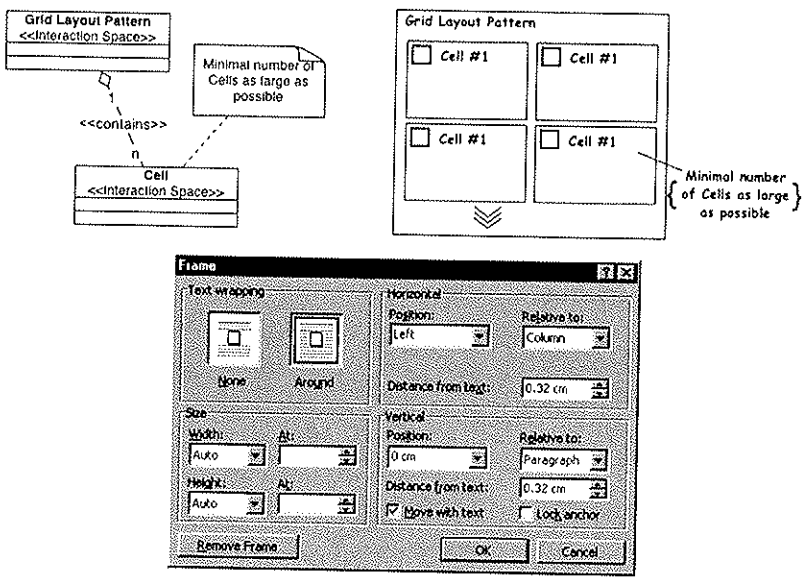


Figure 5.10 The grid layout pattern: a Canonical (top left) and Wisdom (top right) representation and a concrete GUI application (bottom).

It is clear that the Canonical notation has potential for easily expressing patterns that employ spatial, layout or positioning relationships between UI elements. Both notations have mechanisms for adding useful comments and constraints. The repetition element in the Canonical notation (represented by a triple chevron) is expressed as a one-to-many «contains» association in WISDOM.

Figure 5.11 shows a UI pattern where one can see the advantage of Wisdom over CAP. The "Wizard" pattern solves the problem of a user that wants to achieve a single goal, but needs to make several decisions before the goal can be achieved completely, which may not be know to the user (Welie and Trætteberg, 2000). Figure 5.11 shows an instantiation of this pattern through a WISDOM model (top left) that has two interaction spaces: Wizard body and Wizard step. Multiple steps are denoted by the 1..* cardinality in the «contains» association stereotype. Abstract actions (denoted by the «action» operation stereotype) are associated with each interaction space denoting typical actions performed in a Wizard pattern (for instance next, back, cancel and finish) (Nunes, 2003).

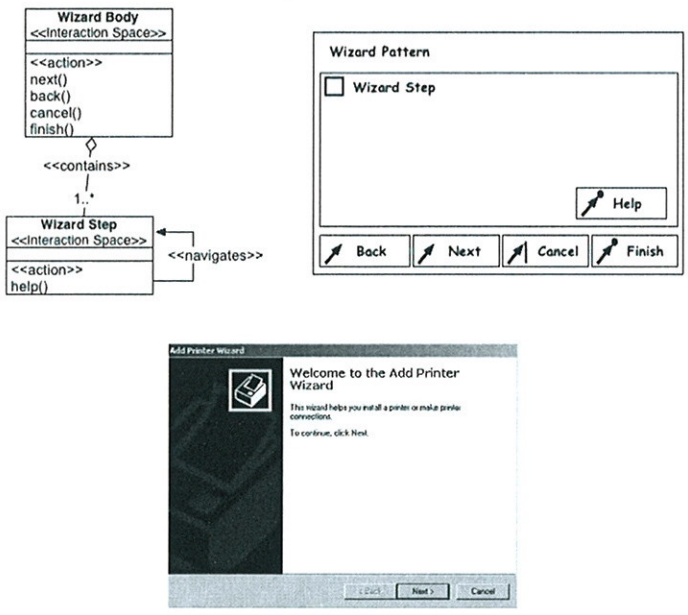


Figure 5.11 The "Wizard" pattern. The top left part of the figure shows the Wisdom UML representation, which shows the navigation between "Wizard steps". The top right shows the Canonical representation and at the bottom a particular realization: the Add Printer Wizard in Windows 2000.

This example illustrates an advantage of WISDOM over CAP regarding the modeling of navigation relationships between the abstract interface elements. In CAP, it is not possible to model a container that allows navigation to other instances of itself (like the Wizard step in this example). Modeling a containment relationship (like a Wizard body that contains successive interaction Wizard steps) is also difficult, unless an informal annotation or comments are used.

Finally, we show yet another abstract design pattern, the Container Navigation pattern (Nilsson, 2002). When the user needs to find an item in a collection of containers, this pattern splits a window into three panes: one for viewing a collection of containers, one for viewing a container and one for viewing individual items. Figure 5.12 shows a WISDOM UML model, the Canonical prototype and a concrete GUI example of this pattern (Netscape's mail/news viewer).

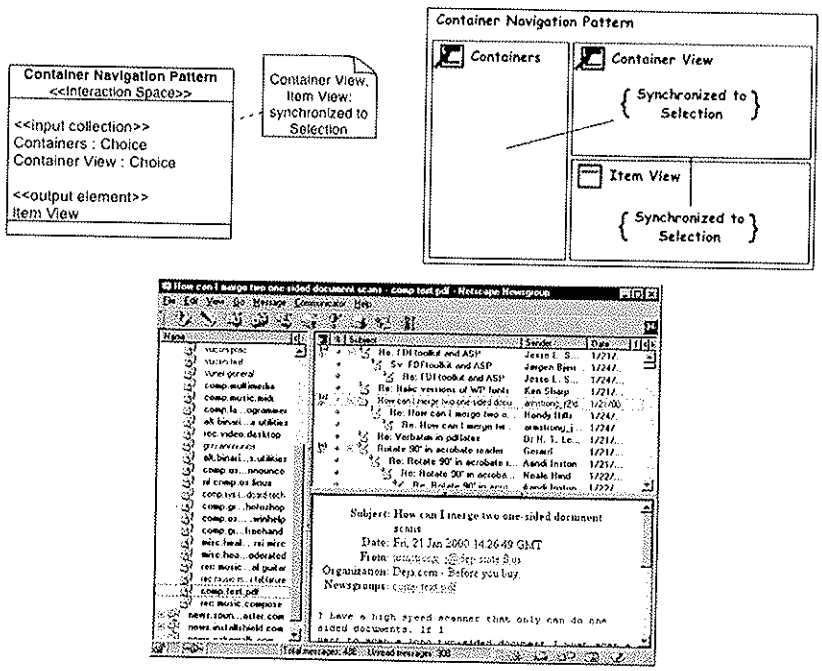


Figure 5.12 The container navigation pattern: a Wisdom (top left) model, a Canonical prototype (top right) and a concrete GUI application (bottom), in this case Netscape's news reader.

In order to adequately express this UI pattern, size and relative positioning do matter. They support the user's task because the user first selects a container, then selects the item

in the container and finally browses through the selected item. The information that the collection of containers occupies the left part of the screen, and that the item view is at the bottom right can only be conveyed through the Canonical notation.

To conclude, we observe that the WISDOM notation has some advantages over CAP, mainly due to its compactness and the fact that is based on a language (UML) well understood and adopted by the majority of software developers and designers. For expressing navigation patterns that involve several interaction spaces, such as the Wizard pattern (Nunes, 2003), the Wisdom notation is more expressive and intuitive. Patterns dealing with spatial layout and size aspects are more clearly represented using CAP. The designer's mind works at several levels of abstraction, thus there is a need for languages and tools supporting those multiple levels of abstraction, while also maintaining a low learning curve.

When trying to express and compare the abstract design patterns presented in this section, we found CanonSketch to be a very useful and practical tool, because it supports two different notations that employ different levels of abstraction and also because it can easily be used to compile a collection of design patterns, thus simplifying the design's comparison and communication.

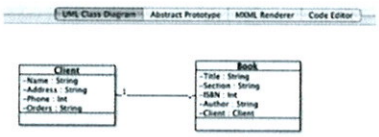
5.2.5 *Final version of CanonSketch*

After the initial version of CanonSketch, and also taking into account several observations, remarks, comments and suggestions given by several users of the tool during informal usability tests, CanonSketch was redesigned.

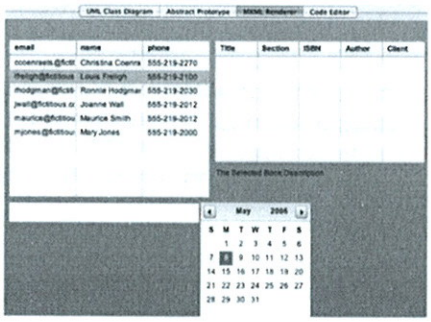
In a first step, the tool's UI was slightly changed, then having the Workstyle Model as inspiration as discussion tool, a new version was built, specifically aimed at better supporting Workstyle transitions (in particular the transitions regarded by professionals as being more important, frequent and difficult).

The final version of CanonSketch (Campos and Nunes, 2006b) shows the user three synchronized views of the UI at different levels of Detail: the UML view (shows the abstract contents of the UI as well as navigation and containment relationships between them); the Canonical Abstract Prototype view (shows the spatial layout and interactive function of the abstract UI elements); and the concrete view (shows a possible concrete

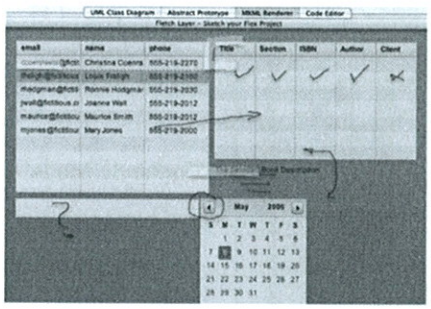
rendering of the UI). There is also a fourth view, the code editor, where the user can edit the textual description of the UI. Figure 5.13 shows an overall illustration of these views. Perspective transitions are supported when the user moves from the UML domain modeling view (problem perspective) to the next views (at design and implementation perspectives). Functionality is added in the code editor view, and can be tested using the concrete view.



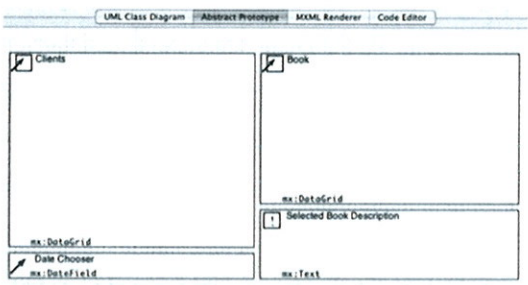
The UML Class View (for Domain Modeling).



The Concrete UI View (for testing the fully functional system).



A layer over the Concrete UI View (for commenting and annotating informally).



The Abstract Prototype View (for laying out the UI, connecting UI elements to UML classes, specifying the concrete UI elements and functions).

The Code Editor View (for adding functionality).

Figure 5.13 A summary of all the views in the current version of CanonSketch [taken from (Campos and Nunes, 2006b)].

CanonSketch supports both informal and formal workstyles. It is possible to use the tool in conjunction with a Tablet or a Smartboard, and apply sketch recognition to informally-drawn sketches (thus transitioning from a semantic-less workstyle to a semantically sound, UML-based model). There is also a semi-transparent layer (shown in the bottom left of Figure 5.13), where the user can freely annotate any of its models or views.

5.2.6 *Generation of MXML*

In this section, we will present the technical details regarding the generation process of MXML code from the Abstract Prototype. This also shows how functionality can be supported easily in a design tool. The other purpose of making this relationship explicit is to help others reproduce our results or to help others adapt these results to similar research contexts or goals.

Figure 5.14 states the possible MXML elements that can be chosen by the designer when he is refining the Abstract Prototype elements. Throughout the three tables in this section, an empty entry for the MXML elements column should be regarded as being the same as the first row (which contains the more generic action, material or hybrid abstract component and its MXML possible implementations).

We can see that the most generic abstract tool is also (possibly) implemented by a very generic MXML tag: mx:Button, which is the MXML equivalent to a common UI button. CheckBoxes, ComboBoxes, DateChoosers, RadioButtons and Lists are all examples of "select" abstract tools. A toggle component would be typically implemented as a CheckBox, although there could be other possible implementations.

These relationships were heavily inspired and guided by the examples provided by Constantine (2003) which can be consulted in page 29 of this thesis (Figure 2.6, Figure 2.7 and Figure 2.8).

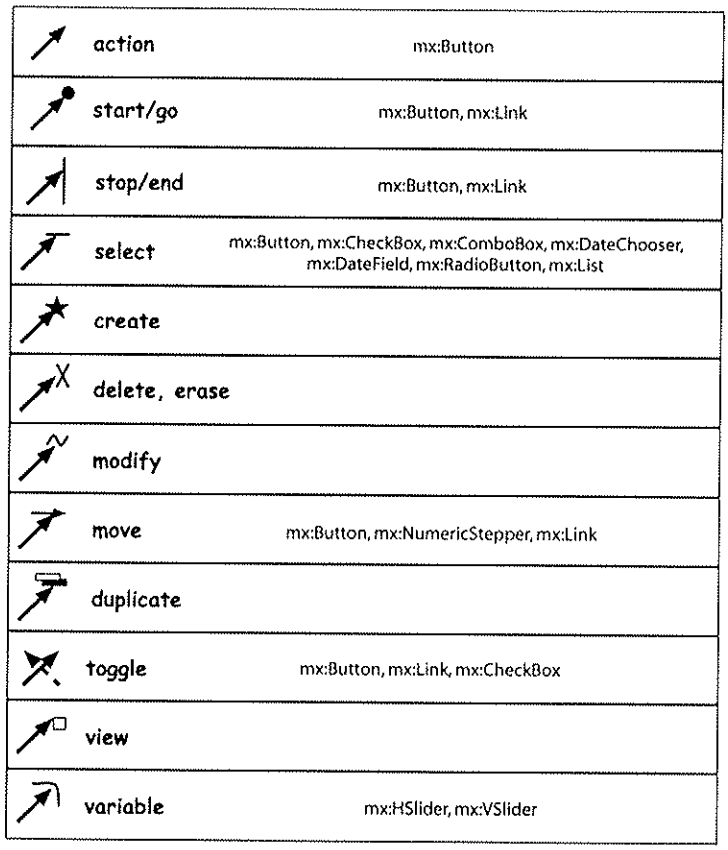


Figure 5.14 Correspondence between all the Canonical Abstract Tools and MXML possible tags for implementing in concrete those abstract tools.

Figure 5.14 shows the same relationship but for abstract materials (abstract elements that only display information to the user, information which cannot be manipulated or changed). Common examples of containers in MXML include Canvas, Panel, Forms, HBoxes and VBoxes. Note that mx:ProgressBar is a good example of a notification component.

Figure 5.15 states possible implementations for active materials (also called hybrids), i.e. abstract components which present information that can also be edited by the user. Text Input boxes, Date Fields and Combo Boxes exemplify concrete active materials. Note that MXML is one of the few languages which implements Constantine's repetition component (in the bottom of Figure 5.15), through a tag called mx:Repeater, very useful when it comes to assembling a repetition pattern such as a mobile phone's pictures list.

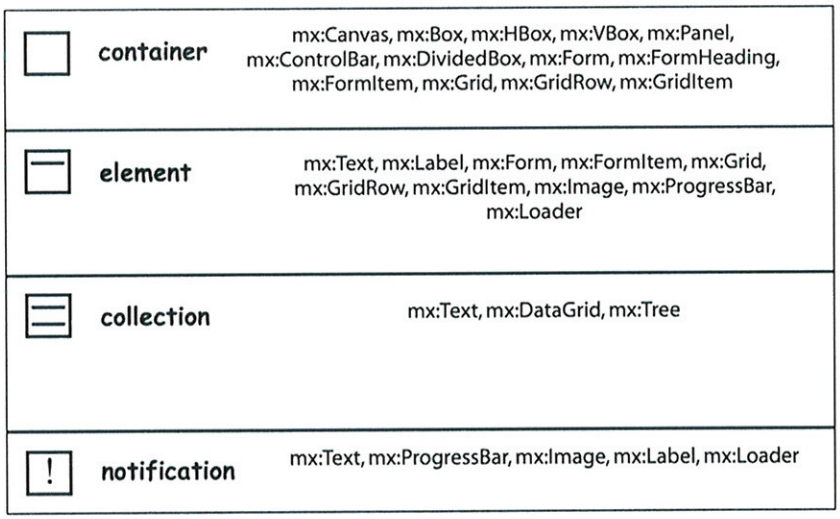


Figure 5.15 Relationship between abstract materials and possible MXML implementations.

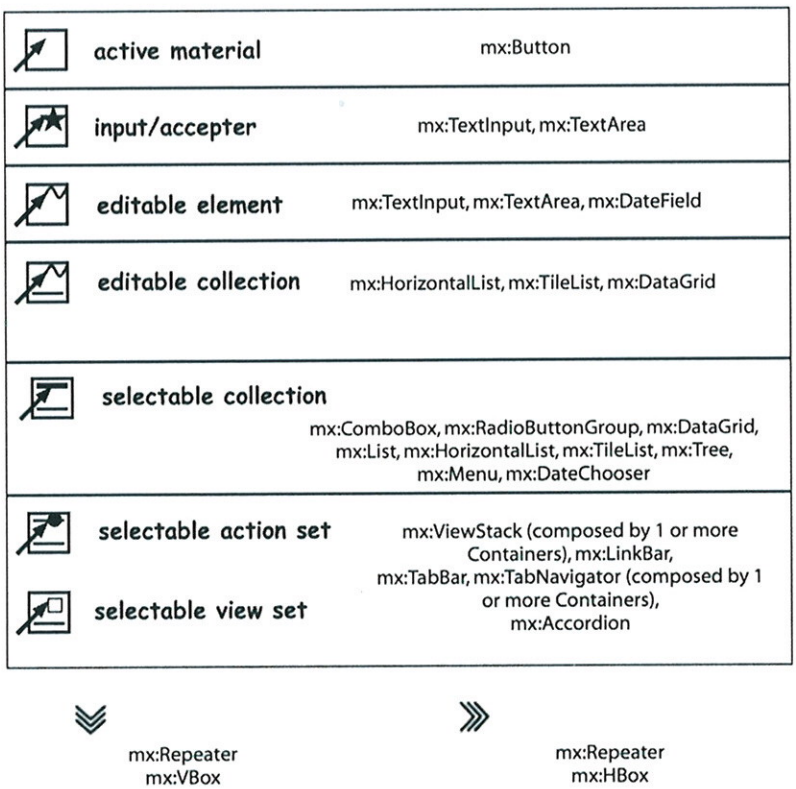


Figure 5.16 Relationship between abstract active materials and possible MXML tags implementing those abstract active materials.

In the final version of CanonSketch, which is illustrated in the next subsection, the generation of MXML is straightforward from either the code view (where it merely compiles the content of the code window) or from the Abstract Prototype view. In this case, the code generated is composed of a set of common lines, specifying namespaces, and for each canonical abstract component:

(i) if the component has already an associated concrete implementation (selected by the designer according to the correspondence presented in Figures 5.14-16) then the tag generated is composed of the MXML name, plus the *x*, *y*, *width* and *height* values of the canonical abstract component; if the tag has a *label*, then the text of the canonical component is attributed to the *label*.

(ii) otherwise, the tag generated is the first tag according to the correspondence tables, plus the *x*, *y*, *width* and *height* values of the canonical abstract component; if the tag has a *label*, then the text of the canonical component is attributed to the *label*.

For instance, if the canonical prototype showed the following model:

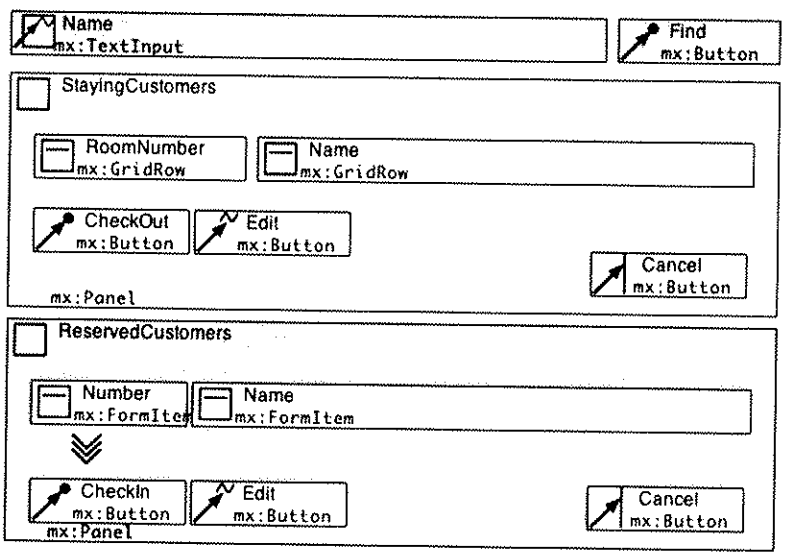


Figure 5.17 An example of a Canonical abstract Prototype annotated with possible concrete implementations.

Then the corresponding generated code would be:

```
<?xml version="1.0" encoding="UTF-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
            styleName="default" fontSize="12" fontFamily="Arial">
    <mx:TextInput x="40" y="48" width="384" height="28">
    </mx:TextInput>
    <mx:Button x="432" y="48" width="104" height="28" label="Find">
    </mx:Button>
    <mx:Panel x="40" y="88" width="496" height="152">
            <mx:FormItem x="56" y="288" width="100" height="28">
            </mx:FormItem>
            <mx:FormItem x="160" y="288" width="360" height="28">
            </mx:FormItem>
            <mx:Button x="56" y="352" width="100" height="28" label="CheckIn">
            </mx:Button>
            <mx:Button x="160" y="352" width="100" height="28" label="Edit">
            </mx:Button>
            <mx:Button x="416" y="352" width="100" height="28" label="Cancel">
            </mx:Button>
    </mx:Panel>
    <mx:Panel x="40" y="248" width="496" height="144">
            <mx:GridRow x="56" y="128" width="136" height="28">
            </mx:GridRow>
            <mx:GridRow x="200" y="128" width="320" height="28">
            </mx:GridRow>
            <mx:Button x="56" y="176" width="100" height="28" label="CheckOut">
            </mx:Button>
            <mx:Button x="160" y="176" width="100" height="28" label="Edit">
            </mx:Button>
            <mx:Button x="416" y="200" width="100" height="28" label="Cancel">
            </mx:Button>
    </mx:Panel>
</mx:Application>
```

Automatic code generation issues and techniques are outside the scope of this thesis. There has been much research aimed at building modeling tools with code generation capabilities. For the case of UI design, the work of Molina (2003), Mori et al., (2004) and Budinsky et al. (1996) are among some of the many proposed approaches. The main interest in this thesis was simply to design and test ideas for supporting the smooth addition of functionality during the complicated work of building an interactive system. There has been much research along these lines, and hopefully this is just a starting point for an interesting journey.

5.2.7 Supporting Workstyle Transitions in CanonSketch

In this section, we will describe how exactly did the workstyle transitions model influenced the design of CanonSketch. Using simple examples side-by-side with the model's pictorial

description, it is possible not only to describe the workstyle transitions supported by a given feature (or set of features) of the tool, but also to evaluate, justify and reflect about support for workstyle transitions.

Consider the example in Figure 5.18. When the designer is modeling the domain concepts using the UML view and then switches to the Abstract Prototype view for laying out the abstract UI elements, the workstyle transition dimensions that change values are Perspective, Formality and Detail, as the Figure illustrates.

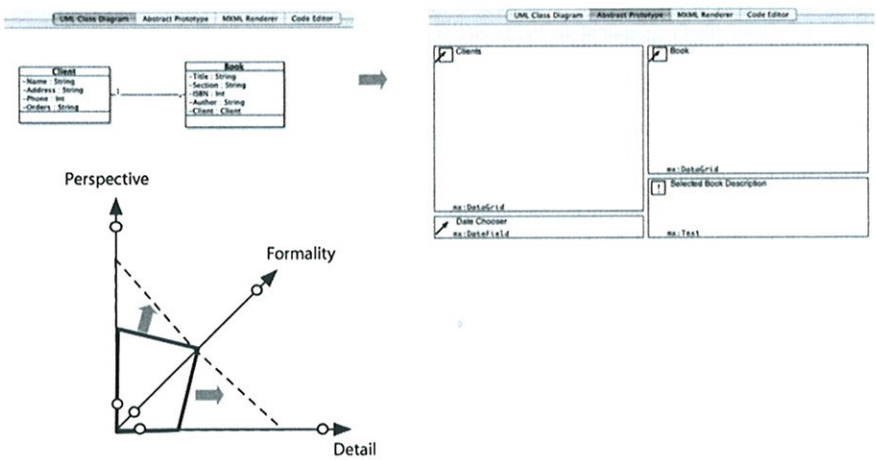


Figure 5.18 The transition supported when the user switches between the UML domain model view and the Abstract Prototype view.

The transition supported by this easy, synchronized switching of views is a transition from a medium value of Perspective into a higher value (but exactly at Solution perspective); and from a low level of detail into a more concrete value of Detail. Figure 5.18 shows the initial workstyle plotted as a thick, solid line, and the final workstyle as a dashed, thinner line. The region between the two is therefore the region supported by this feature of CanonSketch.

Figure 5.19 plots the workstyle transition supported by CanonSketch's switching between Abstract, Non-Functional prototype view and the Concrete, Fully-Functional view. The initial workstyle is characterized as having intermediate values of Perspective, Formality, Detail, Stability and, most importantly, Functionality. Traceability, and the Collaboration-style dimensions Asynchrony and Distribution are irrelevant in describing this transi-

tion. According to our survey's results, this is one of the workstyle transitions considered by professionals as most difficult. Therefore, special care should be given to designing support features for this particular workstyle transition.

A similar workstyle transition, also related with moving from non-functional to fully-functional prototypes, is illustrated in Figure 5.20. Judging from the pictorial description in the left side of the Figure, the surface described is small. However, the transition is difficult: designers have already defined the concrete UI, and have to add the required functionality in order to build a testable UI prototype. In CanonSketch, the MXML markup code is automatically generated from the Abstract Prototype, and using the code editor view, methods, variables and services are added smoothly to the code, thus describing both the visual and the behavior interaction details of the system being designed.

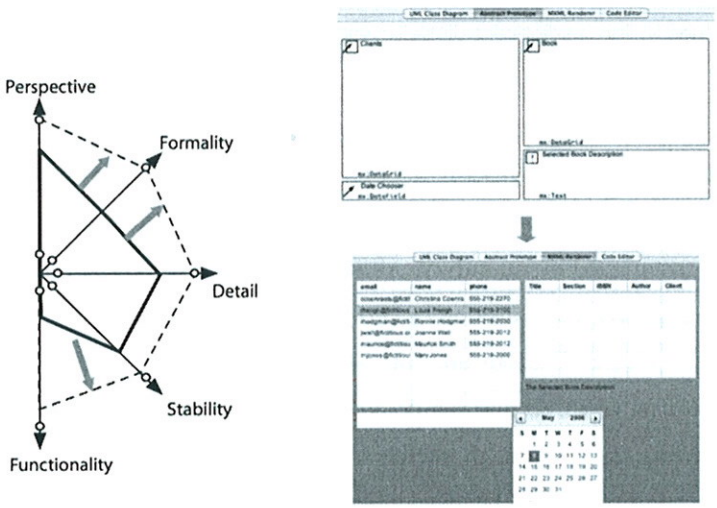


Figure 5.19 Workstyle transition (left) supported by CanonSketch's switching between Abstract, Non-Functional UI view and the Concrete, Fully-Functional view.

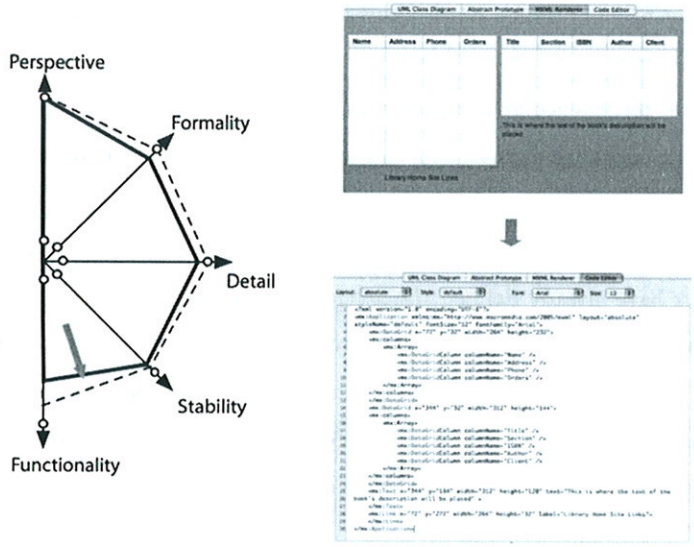


Figure 5.20 A small but significantly difficult transition: moving from a Non-Functional to a Fully-Functional prototype, using the code editor view to add behavior .

As a final example, consider the workstyle transition presented in the left side of Figure 5.21. In this transition, all values remain the same except for the Formality dimension, where a significant change occurs. This is the "Moving from Informal to Formal UI descriptions" as mentioned in our survey's description (see Chapter 4, § 4.6, on page 148). CanonSketch supports this transition by providing the designer with a layer which can be turned on or off via a toolbar button. This layer is put over any model view (including the Concrete prototype view), so that the designer can freely add extra meaning to elements, adding comments, strokes, rough sketches, suggestions and comments/highlights. The best way to achieve this would be by coupling CanonSketch with a SmartBoard, digitizing tablet or similar, sketch-based device.

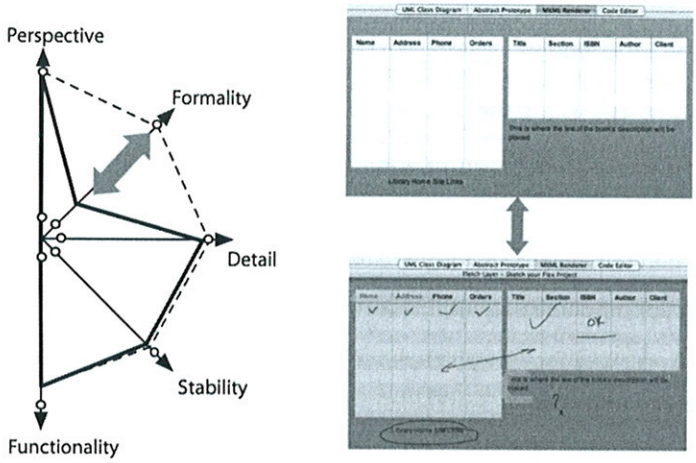


Figure 5.21 Workstyle transition (left) supported by the layer where the designer can freely annotate and comment with rough, free sketches any view of the application (shown is the annotated concrete UI view).

Although not shown in the workstyle transition plot of Figure 5.21, this feature also helps support Collaboration-intensive workstyles, since it fosters communication between members of the design and development team. Newman and Landay (2000), as well as other researchers have shown how both designers and developers recur to sketching over a UI concrete screen (or any other model view) to communicate design issues, particular concerns and design ideas.

5.2.8 A Complete Example: Blog Reader

Simply describing, contextualizing and illustrating isolated examples of workstyle transitions does not reflect the continuous, fluid experience of using a tool that adapts well to workstyle transitions. In order to provide a better understanding of the whole development tasks and how they interconnect and relate, a complete example of building a simple application using CanonSketch is now described.

This example application is quite simple, but covers many of the features and transitions that can be faced by the designer when using a tool like CanonSketch. The Blog Reader is an application for browsing and reading blog entries. It loads RSS feeds by performing an HTTP Service call into a Data Grid, which allows the browsing and selection of a blog entry. The UI design pattern is therefore closely related to the Preview pattern (see Section 5.2.4, page 180), since the user's focus is initially set on the Data Grid (which

should therefore be placed on top of the screen) and afterwards, it is switched to a Text Area where the content of the Blog Entry is displayed (the text area should therefore be placed underneath the data grid).

In the same way as they do with traditional applications and approaches, CanonSketch architects and developers can follow these steps for designing an interactive system:

- Model and implement an adequate object model for the application;
- Choose appropriate relationships between objects (inheritance vs. association, etc...);
- Identify, design and implement applicable design patterns;
- Define and implement a suitable messaging scheme between local objects;
- Define and implement the appropriate infrastructure to communicate with remote services;
- Define the UI layout and interactive functions;
- Choose appropriate implementations for the UI elements;
- Add behavior and full functionality;
- And finally, use the Concrete View to "feel" the proposed solution, by testing and evaluating the Fully-Functional prototype.

This process, however, is iterative, and designers often transition from modeling the objects of the application to defining the UI layout, then going back again to the object model, etc. With a tool like CanonSketch, designed to smooth these transitions, there is support for going through any desired workflow through the steps we mentioned.

Figure 5.22, shows the CanonSketch-made UML Class diagram which describes this example application. The pattern applied is the Model-View-Controller using the WISDOM extensions to the UML language. In this example application, the Model is comprised of only one class, which the designer called "BlogEntry". It has two attributes: Title and Date. In a later phase, the designer added the Content as an HTML text attribute which is handled by the application's Controller, called "BlogService". Finally, the View part of the model is defined as an «Interaction Space» class called "BlogView". This stereotyped class contains the Wisdom attribute stereotypes "Title/Date" (modeled as an «input collection») and "Entry" (modeled as an «output collection»). There's also a stereotyped operation called Load Entries(), modeled as an «action».

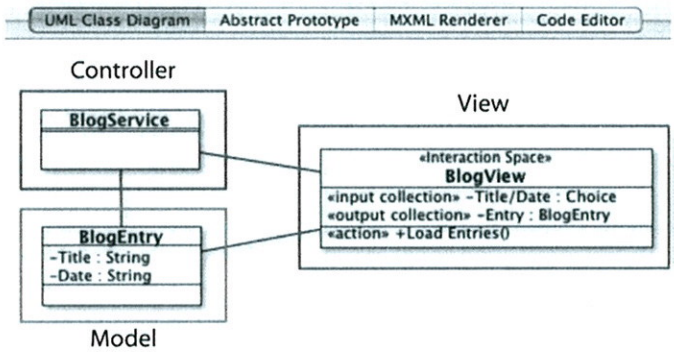


Figure 5.22 Screenshot from CanonSketch showing the application's UML model, using the MVC pattern for structuring the model. Rectangles with the annotations "Controller", "Model" and "View" were added to the screenshot.

From the Interaction Space class, the CanonSketch tool automatically generates the Canonical Abstract Prototype elements following the mapping scheme defined in Figure 5.14 (see page 188). The elements are automatically placed with a default *x-y* coordinates, width and height and the designer just needs to consider the spatial layout aspects and define the desired elements' positions and sizes (this is another idea that comes from supporting detail transitions in our model). For instance, the BlogView class in Figure 5.22 will automatically generate a new interaction space containing a selectable collection named "Title/Date", a collection named "BlogEntry" and an generic action/operation named "Load Entries". These components are automatically placed in the CAP view, with non-overlapping default *x-y* coordinates, width and height. From then on, the designer can manipulate them and CanonSketch maintains coherence between the UML and CAP views. If the designer changes the "Blog Entry" attribute in the Blog View class (using the UML view) from output collection to input collection, then the corresponding CAP will change from a canonical abstract collection into an abstract active material. Conversely, a change of name or interactive function in any CAP element will be reflected immediately in the UML view. This is accomplished thanks to the embedded UML metamodel and the mapping scheme described in §5.2.3.

However, this would not be a mandatory workflow. The designer could have chosen to adopt a more informal workstyle, switching to the Abstract Prototype view and freely sketching a rough wireframe which would be the skeleton of the application's screen, as Figure 5.23 shows. If the designer chooses to define this wireframe first, then he or she

could apply the sketch recognition engine and automatically and effortlessly introduce formality and detail to the wireframe, transitioning from an informal, meaningless sketch of the UI's layout into a more formal, semantically-sound (and based on the UML) Canonical Abstract Prototype.

Performing this sketch recognition (accomplished thanks to the CALI library (Caetano et al., 2002), which we integrated with Mac OS X's Objective-C), the resulting model would be similar to the one shown in Figure 5.25.

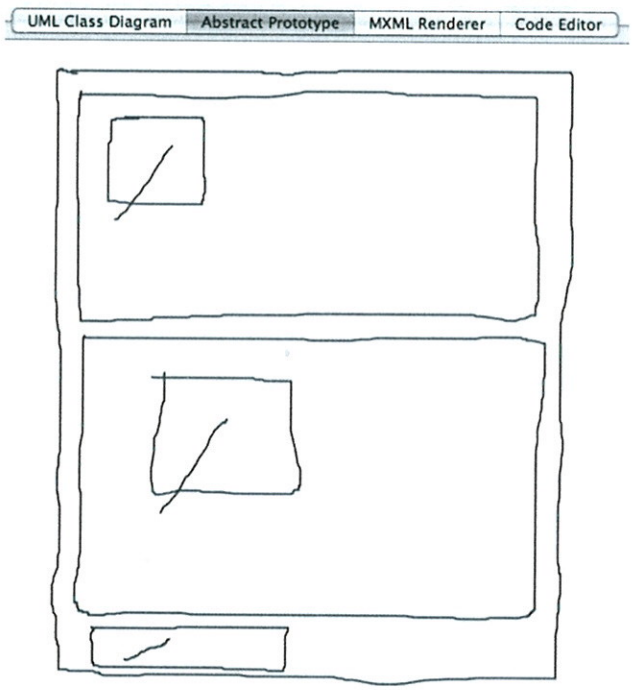


Figure 5.23 Rough sketch of the application's wireframe scheme, before applying the sketch recognition algorithm.

To enable a fast creation of CAPs using sketch recognition, the following scheme was implemented thanks to the CALI library, which comes with the useful feature of allowing recognition of composed shapes:

- If the sketch is recognized as a square shape, and if it contains other sketches, then an abstract container is generated, and its contained elements will be generated from the application of the sketch recognition to each of them;

- If the shape is recognized as a square shape, and it only contains a single stroke, then it is recognized as an abstract action/operation (see the leftmost example in Figure 5.24);
- If the shape is recognized as a square shape and it only contains a square shape then it is recognized as an abstract material (see the center of Figure 5.24);
- If the shape is recognized as a square shape and it contains a composed shape with a square and an intersecting stroke, then it is recognized as an abstract active material (this is the case of the rightmost example in Figure 5.24).

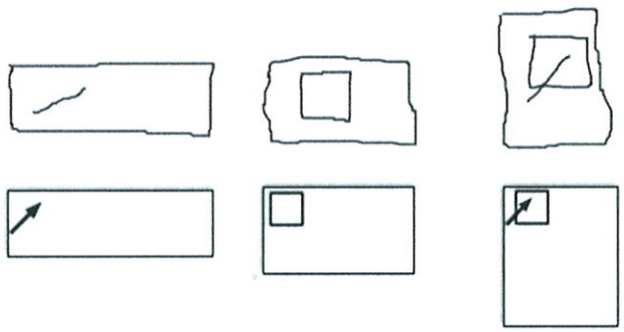


Figure 5.24 Example shapes (on top) and the CAP elements that are automatically recognized.

The particular Canonical Abstract Prototype shown in Figure 5.25 is already annotated with the concrete MXML elements, chosen by the designer, which will implement each of the abstract elements. In this case, a container (implemented as an mx:Panel) will enclose the elements of the application: the 'Title/Date" active material which shows information upon which the user can act or manipulate (and was originated from the UML Model class which was called "BlogEntry"), the "Text Entry", also an active material and the "Load entries" action.

The designer opted for turning the 'Title/Date" active material into a concrete element as a "mx:DataGrid", a commonly used component that uses a Table UI to display

collections of elements. "Text Entry" was given a "mx:TextArea" tag, and "Load Entries" was concretized as a simple "mx:Button"[4].

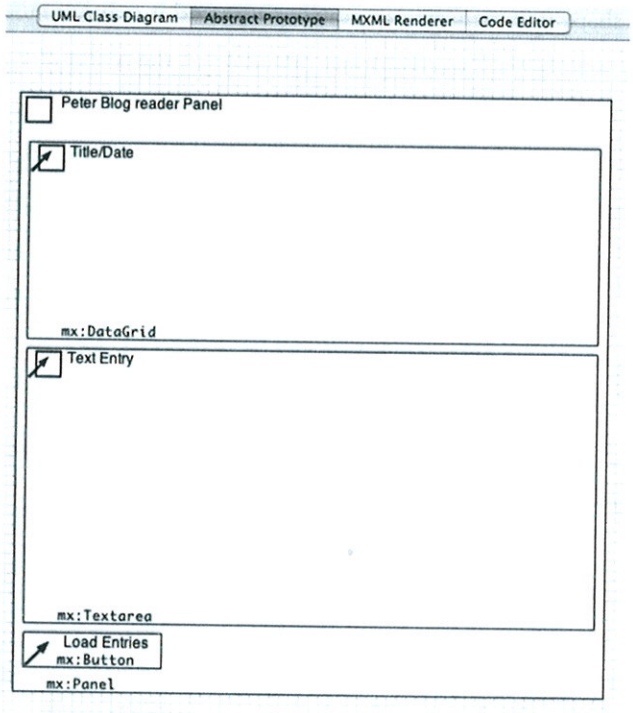


Figure 5.25 Abstract Prototype after being refined from the sketch-recognized model. The lower part of the elements shows the concrete MXML representation, as chosen by the designer.

CanonSketch provides a contextual menu for any abstract element, where the designer chooses the concrete MXML element, using the correspondence of Canonical Elements – possible MXML concrete realization, which was presented in § 5.2.6. It is reasonable to state, that CanonSketch provides partial automation, lying in between the two extremes of informal tools such as Silk or Denim (Landay and Myers, 2001), which are not focused on generating code, and the fully-oriented to code-generation tools such as OlivaNova Execution System (Molina, 2003). The aim of this is to provide the best of both worlds and more

[4] For more information about the MXML language, please refer to Section 3.6.6, "Adobe MXML and Flex" or to Adobe's official MXML website: http://www.adobe.com/devnet/flex.

importantly, to let the designer choose his preferred workstyle, concentrating on facilitating smooth transitions between workstyles.

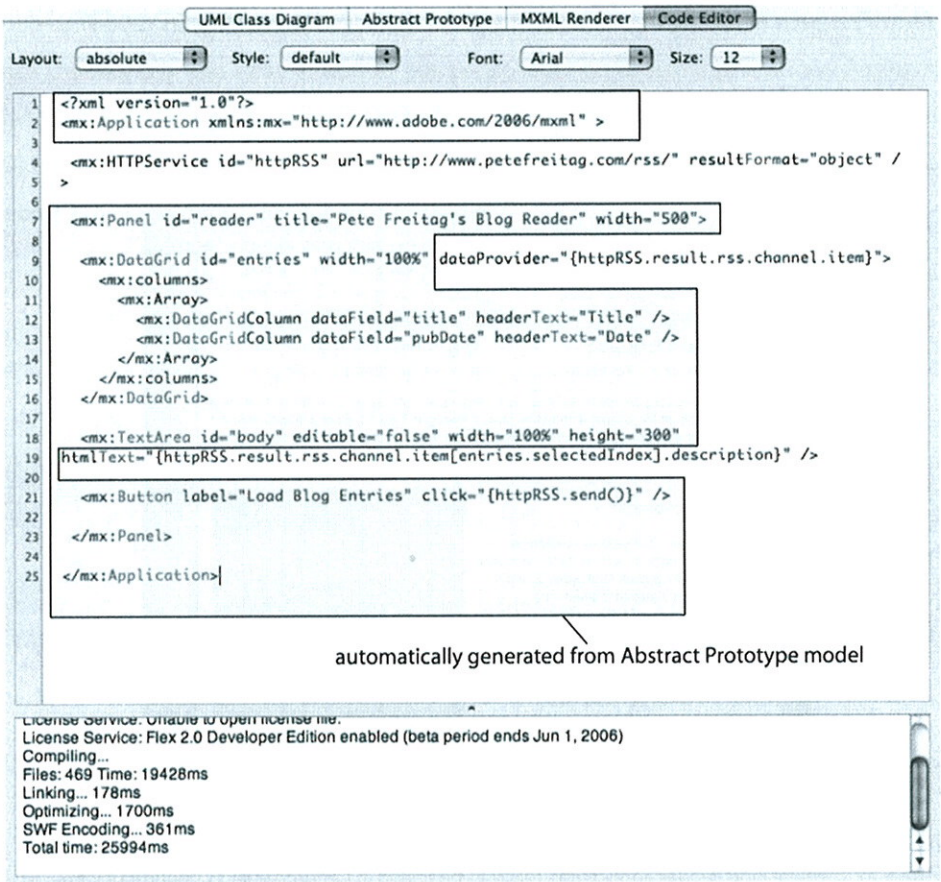


Figure 5.26 The complete source code of the application. Highlighted is the code automatically generated by CanonSketch from the Abstract Prototype model.

Figure 5.26 presents the corresponding Code Editor view for this Blog Reader application. Highlighted are the code segments which were automatically generated by CanonSketch from the Canonical Abstract Prototype. The designer only had to code the HTTP service which loads the data from the Blog, specify the HTML text which should be displayed in the Text Area component, and state the data provider of the Entries' Data Grid (the result of the HTTP Service call). The bottom part of the Figure also shows the feedback panel which provides information regarding the state of the compilation process.

CanonSketch offers syntax highlighting, line numbering and other features one might expect from a common Project Builder. However, and since it is basically a proof-of-concept tool, it still lacks more code-editing supporting techniques, something that could be part of future research following e.g. some of the research lines of Ko and Myers (2003).

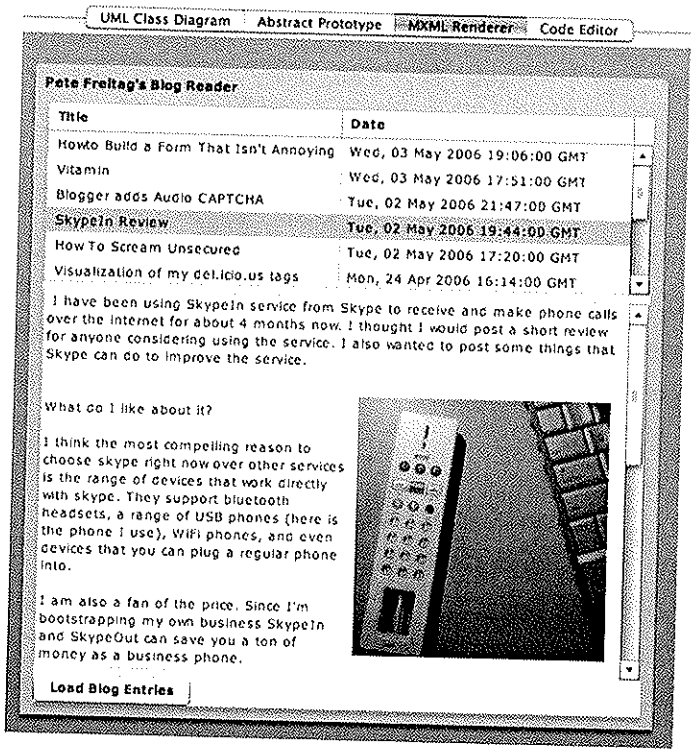


Figure 5.27 The final Blog Reader application, ready to be tested in CanonSketch's Concrete UI view.

Last, but not least, Figure 5.27 shows the final Blog Reader application, showing a blog entry's Title and Date selected and the corresponding text and images of the entry.

Although this is a simple example, provided with the goal of illustrating as briefly as possible all the relevant aspects present in the tool, complex applications such as Art Tickets Selling Systems or Online Mobile Phones Stores can be specified and even built using CanonSketch. In fact, and apart some implementation issues that are missing, debugging is the only activity which is not handled by the tool, although many issues for improvement can be found and stated (for a more detailed discussion of the limitations of the approach and tools, please refer to CHAPTER 7, § 7.1). HCI students at the University of

Madeira used an intermediate version of the CanonSketch tool, together with a Java UI Builder to specify and design a more complex application. CanonSketch has also been used in several other universities around the world, mainly for UML modeling but also for teaching UsageCD concepts. A commercial version is foreseeable and is being undertaken, since the tool has also clear industrial potential. Some professional practitioners using the Mac platform use it to model UML domain models. The final, MXML-supporting version will only be released at the time of this thesis writing, so an increase of interest in the tool among MXML and Flex developers is expected.

5.3 The TaskSketch Tool

The results of the survey presented in CHAPTER 4.6 (and in particular §4.6.4) have clear implications for developers of software design tools that incorporate support to UI-related activities, in particular analysis-related activities.

Of all the transitions we mentioned in the form of scenarios, "*Moving from problem space concepts into final solution design and back*" was considered both the most difficult and the most frequent transition (see Figure 4.26 on page 57). As an example of how this information could be exploited, consider one idea for supporting changes in perspective (problem space or solution space) we have been exploring in prototype tools: imagine a development environment where clicking on a use case could take the developer to its definition in a glossary. Selecting that use case could also show the components that support it, or the concrete widgets for a given realization of that model. Even entries in help files could be linked to the user roles they support, or to the actual code and visual UI controls. This idea has long been evangelized by Larry Constantine, who has provided us with many ideas we are now trying to materialize in the form of testable tools (Constantine and Campos, 2005).

The TaskSketch tool (Campos, 2005a, 2005b) is being developed both as a research proof-of-concept instrument used to test and validate our ideas, as well as an instructive tool, which has proven effective in supporting the teaching of UCD concepts at University level. Figure 5.28 shows a partial screenshot from the tool, illustrating briefly all the views and models that are supported. It shows how the "Moving from problem space concepts into final solution design and back" transition can be better supported. Instead of merely supplying the users with a set of constructs and views for use case modeling, TaskSketch provides drag-and-drop mechanisms for extracting an initial conceptual architecture for an interactive system. The user smoothly changes from a task case view (described as a UML activity diagram) into a system architecture view. By simply using drag-and-drop between views, TaskSketch converts each model element in a view to the corresponding semantic equivalent in the other. This way, TaskSketch helps the system analyst/designer to concentrate on what's really important: architecting the system.

From our experience in the context of small software development companies, traceability support is important, not only to reduce the cognitive load of designers and engineers, but also to help in tasks such as prioritizing and negotiating development tasks. Our tool TaskSketch tries to support this by color-coding elements in relation to their use cases: this way, the most colored elements are immediately seen by the tool's users as supporting more use cases (and this also means that the element, whether it is a user description or a concrete UI widget, will probably be more difficult to implement).

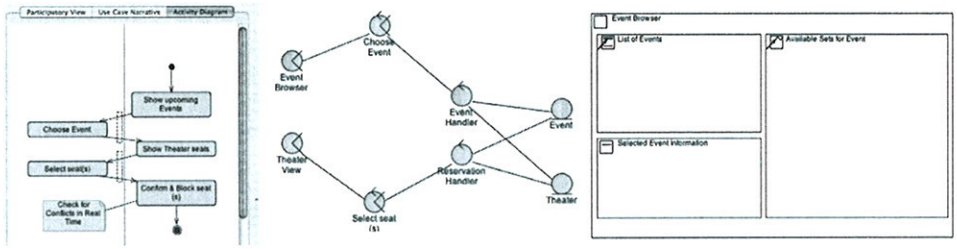


Figure 5.28 The TaskSketch tool showing (from left to right): the UML Activity Diagram View, the System's Conceptual Architecture and the UI Abstract Layout.

Figure 5.28 also shows that the tool also supports editing of Canonical Abstract Prototypes. Although CAPs are at a design level – and TaskSketch is primarily positioned as an Analysis tool – we found that it would help students transition into the Design phase, and although this is not semantically correct, in this implementation of TaskSketch clicking on an Interaction Space (an analysis-level model element) opens up a window where the user can draw CAPs for that Interaction Space. This is also a way of supporting High-Detail to Low-Detail back-and-forth transitions.

5.3.1 Supporting Perspective and Traceability Transitions

TaskSketch focuses on linking and tracing use cases to the conceptual architecture of an IS. The idea is to use the Wisdom extension to the UML (Nunes and Cunha, 2001), which can be summarized as the UML class stereotypes in the lower right part of Figure 5.29:

- «Interaction Space» (models the interaction between the IS and human users within the user interface of that IS);
- «Task» (models 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);

- «Control» (encapsulates complex derivations and calculations, such as business logic, that cannot be related to specific entity classes) and
- «Entity» (models perdurable, often persistent, information).

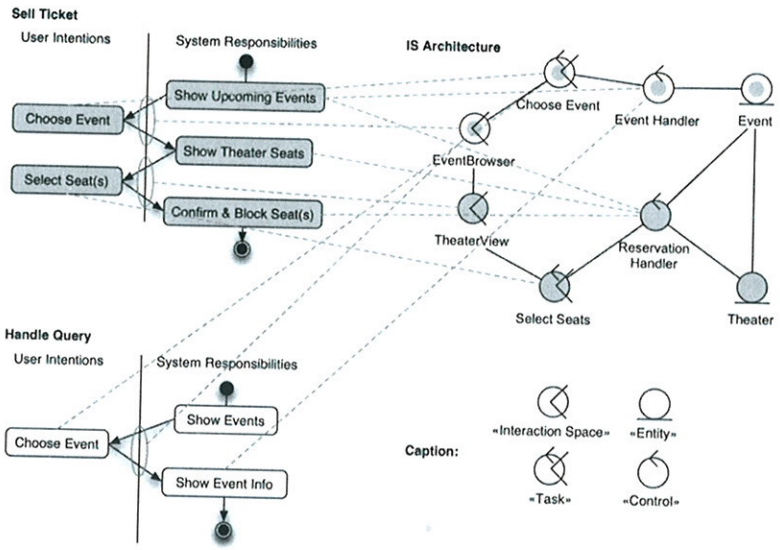


Figure 5.29 Tracing use cases and task activities to the conceptual architecture of an Information System: a simple example of an Arts Center ticket selling IS.

Figure 5.29 shows a simple example of two UML activity diagrams representing task flows of two distinct use cases in an Arts Center ticket selling IS: Sell Ticket (represented in blue) and Handle Query (represented in orange). Each use case is detailed using two swim lanes: User Intentions and System Responsibilities (the tool will also provide a participatory view and a narrative view similar to using an index card as described in (Constantine and Lockwood, 1999)). For example, in the use case "Sell Ticket", it is the system responsibility to show upcoming events so that the user (the ticket selling agent) chooses one of them. The system then shows the available theater seats for that event and the user selects the desired seats, which are blocked by the IS. Each crossing of the swim lane originates an interaction space (Event Browser and Theater View in this example). Each action on the User Intentions swim lane corresponds to a task and each action on the System Responsibilities swim lane is associated with a control.

This extraction scheme of architectural elements from the use cases is a central technique in the Wisdom method which is leveraged in the TaskSketch tool.

Figure 5.29 describes these relations for these two distinct use cases. Using color, the developer can look at the architectural view of the system and see which classes handle which use cases. This simple support to requirements traceability can be very powerful for, e.g., prioritizing development by deciding which classes are more urgent to implement. Figure 5.30 shows a screenshot of the initial version of the tool.

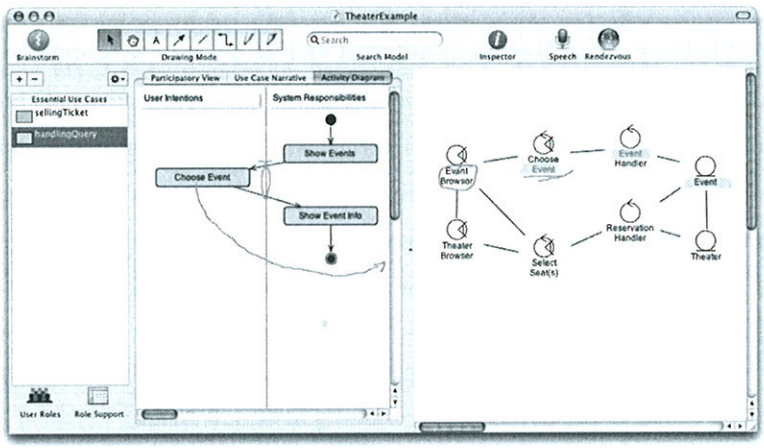


Figure 5.30 Screenshot of the development environment in the initial version of TaskSketch.

As another example of workstyle support, the tool allows editing of task flows at three different – but synchronized – views: the participatory view, which is a typical result of a participatory session with end users (obtained by manipulation of sticky notes); the use case narrative proposed by Usage-Centered Design that can also be printed in index cards for stacking and ordering by the different stakeholders; and the activity diagram which could include additional details relevant to developers but that are not depicted in the other views.

The idea of supporting multiple notations, which are synchronized through a common semantic model, is central to this dissertation. The workstyle model suggests that transitions in notational-styles are needed for current tools, and this TaskSketch example shows how to implement it, in the context of UCD tools.

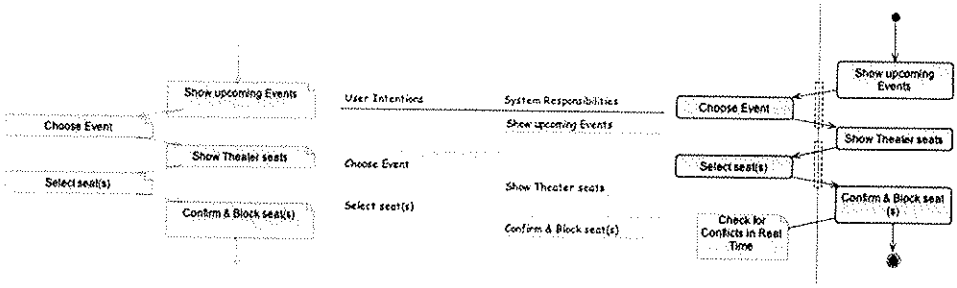


Figure 5.31 The three synchronized views in TaskSketch (from left to right): the participatory view, based on post-it notes, the Usage-centered use case Narrative and the UML-based activity diagram view.

Figure 5.31 shows a TaskSketch screenshot of each of the three synchronized views for the use case "Sell Ticket" mentioned previously. The UML-based activity diagram is oriented to software engineers, particularly concerned with modeling restrictions and semantic issues. The UsageCD use case narrative is particularly interaction design-oriented, clearly more prosaic and embedded in an interface mimicking the index cards employed in UsageCD. The participatory view, based on post-it sticky notes is adequate for collaborative sessions with clients or potential users. The focus here is to allow flexibility of choosing the preferred notation as well as to *transition* between notations fluidly and effortlessly.

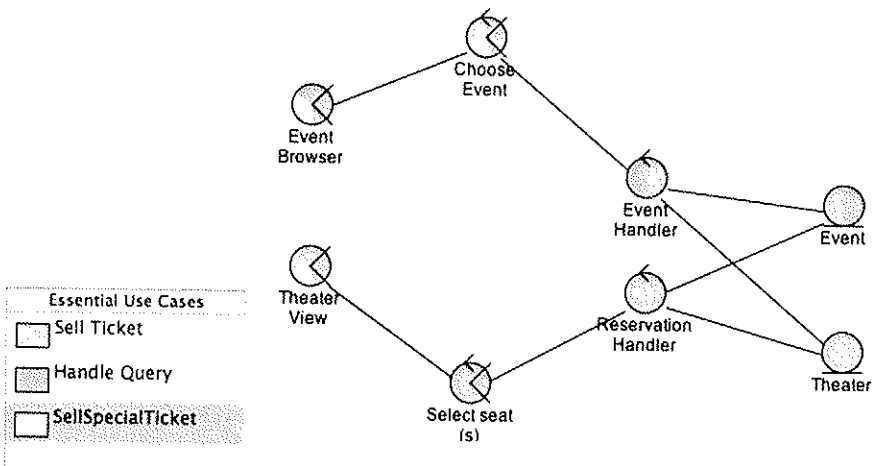


Figure 5.32 Use Cases view (automatically color-coded) and the Conceptual Architecture of the system showing the traceability relationships through the use of color.

Finally, Figure 5.32 shows another partial screenshot from TaskSketch: on the lower left side of the Figure, the Essential Use Cases view provides the user with the title and

color (which is attributed by default) of each of the use cases being modeled in the tool. This example shows three use cases with some similar steps, so after establishing the traceability relationships, the Conceptual architecture of the system (shown on the right of Figure 5.32) has some elements which have more than just one color, meaning they implement more than one use case. Consider for instance, the "Event Browser" interaction space. Since it has all the colors from the use cases, this will probably mean that implementing this particular interaction space will require much more development effort. This idea of color-coding the elements and using color as a traceability mechanism per excellence was in fact informed by observation of real world developers using the WISDOM method, who made the diagrams by hand using paper and many colored pens. In other words, developers who could use any UML-based tool for modeling WISDOM concepts actually went through the effort of doing it manually only to be able to use color-coding.

5.3.2 *Supporting Formality Transitions*

We also concentrated efforts on exploring the possibilities offered by gesture recognition, mixing formal and informal notations and collaborative development using speech recognition and a shared display. Supporting levels of formality is also achieved in TaskSketch: as in CanonSketch, we linked the CALI (Caetano et al., 2002) library of sketch recognition algorithms with the TaskSketch tool, so it is possible to allow a mixture of sketches and then select a few of those semantic-less sketches to be automatically recognized by TaskSketch.

Another issue was gesture recognition. This feature was not evaluated but it was implemented and informally tested: a set of gestures was pre-defined in TaskSketch and for each gesture a meaning was associated. In future work, it might be interesting to study whether or not this modality could be exploited as a way of fostering stakeholders' involvement in the requirements gathering.

As a final manner of supporting Formality transitions, speech recognition is incorporated into the tool. It is possible to define a set of voice commands and then associate each of the voice commands an action or set of actions. The speech recognition system was given an original utility for supporting collaboration, described in the next subsection.

5.3.3 *Supporting Asynchrony and Distribution Transitions*

The best way to have a good idea is to have lots of ideas.
Linus Pauling

There is evidence that real-time collaboration tools incorporating speech recognition and displaying information about a group's dynamics can positively impact the group's interaction (DiMicco et al., 2004).

In some decision tasks, in particular during requirements elicitation and finding the core classes of an IS, there is a risk that some stakeholders holding important information will not effectively share it, thus leading the team to less informed discussions.

In the "Brainstorm Environment" we propose (as part of the TaskSketch tool mentioned above), each stakeholder is associated a color and types in ideas for core classes/concepts or requirements of the IS being developed. Every time someone sends a concept to the screen, a shape color-coded by the user who sent it starts slowly falling through the center of the window.

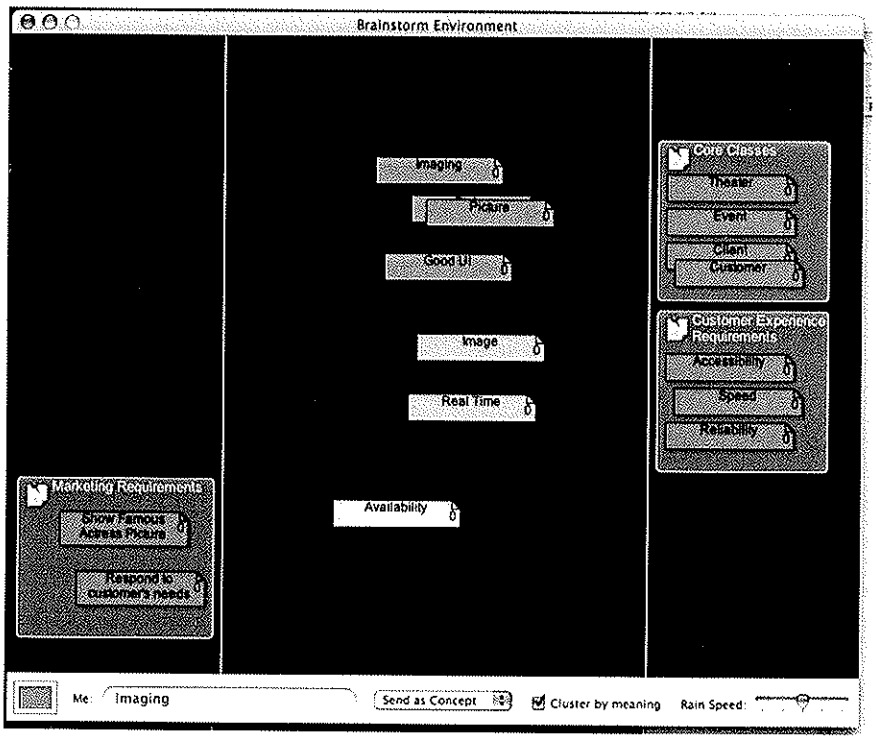


Figure 5.33 Screenshot of the proposed brainstorm environment for collaborative core concepts discussion and clustering.

The content of this shared display can be manipulated by anyone, so it becomes useful to cluster concepts manually. Dragging a shape to the left or right sides of the window makes it stop falling. Concepts that remained untouched become grouped in the bottom of the window, in the same way as in (DiMicco et al., 2004). Clustering of concepts can also be made automatically, because this system uses a thesaurus and every time someone sends a common concept, such as "client" and "customer", the two shapes become aggregated. The speech recognition system is set to dynamically recognize any of the phrases or words in the shared display. Every time a concept is recognized, the shape shows a number, which counts the number of times that concept has been spoken during the meeting. Figure 5.33 shows a screenshot of this environment.

Users can also create "concept bins", which act as context providers for concepts. A close-up shot of concept bins for the Art Tickets selling system is shown in Figure 5.34. The concept bin "Core Classes" was set when the meeting started and domain classes had to be brainstormed. For this example system, the core classes found were "Theater", "Event", "Client" and Customer". The built-in thesaurus system recognized that "Client" and "Customer" were the same concept so it clustered them automatically. Other bins shown in the Figure are "Marketing Requirements" ("Show famous actress picture", "Respond to customer's needs") and "Customer Experience Requirements" ("Availability", "Speed" and "Reliability").

If the user drags the concept bin around the screen, all the contained concepts will also move. The concept bin acts as a place where designers can cluster ideas and concepts. But also as a place where good ideas can be "saved" for future discussions.

Dragging a concept to a bin associates it with that context. Thus, it is possible to set the speech recognition system to recognize concepts only within a given context. In other words, if the meeting starts with a discussion of the core domain classes, the speech recognition system will count the frequency of concepts spoken during the context of the domain classes discussion. If later on the meeting participants change the discussion to another context, such as marketing requirements, the speech recognition system will not account for concepts which were assigned to the previous context discussions.

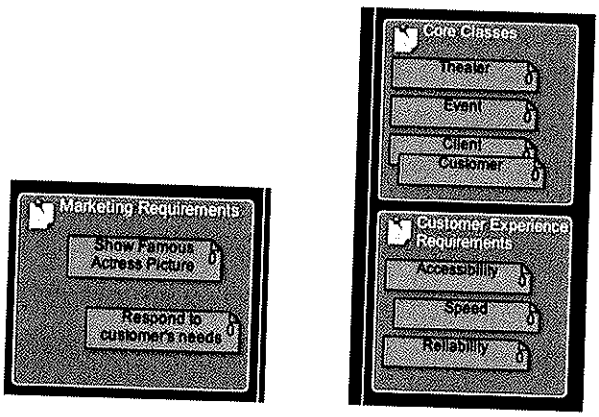


Figure 5.34 A close-up of some concept bins for the Art Tickets selling system, containing items during the discussion of the relevant domain classes, customer experience requirements and marketing requirements.

We foresee the following usage scenario for an environment like this one: different stakeholders meet to discuss and identify the main concepts (or classes) of the IS to develop. This includes clients, marketers, programmers and interaction designers. Each uses a microphone and has its own color. As they suggest ideas, they watch them fall and the display becomes color-filled. Thus, this system attempts to increase the discussion of ideas as well as to foster collaboration between people with different backgrounds through an engaging environment. It is also expected that under-speakers will participate more and over-speakers will participate less, like (DiMicco et al., 2004) have shown. However, by the end of the meeting, it is also expected to achieve a better clustering of concepts as well as to have an idea of what concepts are more important (by looking at which concepts were more referred to during the meeting).

Aiming at achieving a better support for collaboration-related workstyles, we used blogs as common use case repositories fostering offline collaboration (different time, different places). Users can post comments, make changes using web browsers and simply browse the use case narratives, and then designers can choose one or more blog posts and retrieve them into the semantically-sound TaskSketch models. Figure 5.35 illustrates this mechanism.

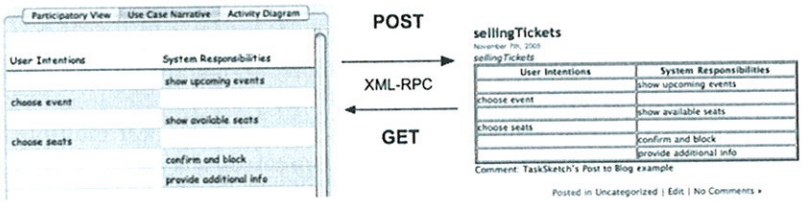


Figure 5.35 The Post and Get blogging mechanism incorporated in TaskSketch.

On the left side of Figure 5.35 we show a screenshot from the Use Case Narrative view in TaskSketch (the particular task case is from the Art Tickets Selling System we referenced earlier). Through an implementation of XML-RPC communication protocols between the blog's APIs (eBlogger and WordPress were the Blog systems tested) and the TaskSketch tool, it is possible to Post a use case narrative to the Blog's web page, and also Get i.e. retrieve a use case narrative from the Blog into the UML-sound TaskSketch format. On the right side of Figure 5.35 we present the corresponding screenshot taken directly from the browser window displaying the blog's web page where users can edit, post comments and simply browse the use case narratives.

5.4 Conclusions

This work shows that the UML semantic model can be used to support multiple levels of detail of an interactive system. The UML had a tremendous impact in software engineering but still remains quite far from achieving the promises of the late 90s. We still lack widely accepted notations to support user-centered development and user-interface design. We think that CanonSketch is a step towards that goal in bridging HCI and Software Engineering. We are attempting to change the way modeling tools are built and envisioned by focusing on achieving a flexible and usable tool instead of following formalism-centered approaches.

To offer software engineers a usable, efficient and effective set of tools and methods is an important step towards building valuable, easy to use software. The same concepts that apply to the production of usable software also apply to the production of modeling tools. The CanonSketch project described here attempts to change the way modeling tools are built and envisioned. Instead of focusing on the mechanisms required for automatic generation techniques, we focus on the successful features of usable software and on interaction idioms more closely related to Office-like applications.

We showed how useful the tool could be in expressing UI patterns, and compared Wisdom UML representations of some patterns to the Canonical representations using the proposed correspondence between the two notations. Patterns dealing with spatial or layout aspects could be adequately expressed in a Canonical representation, while Wisdom UML is better at modeling navigation relationships. With the consequent integration of the semantic model of the UML into the tool we were able, among other possibilities, to export the abstract UI specification in XMI format, thus promoting artifact exchange between UML-based tools.

We were also able to extend CanonSketch in order to better support functionality transitions. Adding a code editor view and the possibility of choosing a concrete implementation for each of the abstract components being modeled meant that we turned CanonSketch into an almost-complete development environment. Debugging capabilities were left aside of our research, but integration with current techniques such as the Whyline (Ko and Myers, 2002) could be promising.

The TaskSketch tool was developed following a similar approach as in CanonSketch. The workstyle-based inspirations were sometimes the same (as in the three use case steps synchronized views, which is nothing but a way to support different levels of detail, perspective and formality) and sometimes different (as the Blog integration mechanism or the Collaborative Brainstorming environment show).

Requirements management and elicitation is widely recognized to be one of the major problems in modern software development. This stage of development involves multiple stakeholders, usually with different backgrounds, and is currently faced with the advent of multi-platform development (Patrício et al., 2004). In this context, new tools are required to enable cooperation and communication of multiple stakeholders over a common semantic model that is capable of driving modern software development. Applying User-centered design to the design of new tools for promoting co-authoring and co-evolutionary development of requirements over a common semantic model brings many benefits, such as increased stakeholder involvement and information sharing, in-creased traceability and usable ways to negotiate requirements as well as prioritize development tasks. The Task-Sketch tool described in this dissertation is a step forward in what regards the usability of current requirements and analysis tools.

6 Evaluating the Tools

Models, Methods and Results

"If something exists, it exists in some amount.
If it exists in some amount, then it is capable of being measured."

Rene Descartes, Principles of Philosophy, 1644

Measuring the usability of any tool is a difficult and intricate task. This chapter is entirely dedicated to presenting and reflecting upon usability methods which were designed and performed with the goal of studying (a) the tools' workstyle support level and (b) the tools' probabilities of being adopted by users.

Through the findings in these studies, we demonstrate in this chapter that the developed tools exhibit a satisfactory level of both usability and usefulness. This supports our initial claim, presented in the problem statement (see §1.2), that "modeling the interaction designers' workstyles and using that information in the design of tools that can efficiently support *transitions* in those workstyles will lead to more useful and usable tools".

This chapter is organized the following way: SECTION 6.1 presents in some detail an evaluation study performed with the CanonSketch tool. The study's design tasks, procedure and results are described. The section ends with a brief summary of the most relevant design findings, which provide new insight regarding the requirements and usage issues for this class of tools.

In SECTION 6.2, a framework for studying the designer's tools and workstyles is described. The framework is basically an extension of the Technology Acceptance Model (TAM) (Davis, 1989) which was described in §2.3.1 of this dissertation. The originality of this approach stems from the fact that it combines quantitative, workstyle-related data with qualitative impressions about the tool's usability and usefulness. By quantitative we mean data such as e.g. the frequency of change of perspective/views during the modeling

session, and by qualitative, we mean the rating (using a 7-point Likert scale) given to the usability, usefulness and intentions to use the tool, as rated by the test users after the modeling session. This new framework served as a test benchmark for assessing the level of acceptance and behavioral intentions of use regarding the TaskSketch tool, described in the previous chapter. We demonstrate that users expressed a positive intention to use the tool, which corroborates one of the proposals of this thesis, and we also study the influence of performing workstyle transitions in the qualitative impressions of use.

SECTION 6.3 is the result of an extensive, 2-year period of observations and studies regarding the tools' usage, and it describes a set of principles and practice which arise from the experiences, observations, feedback and data collected during this research period. The principles and practice of workstyle modeling, as it is called, is focused on stating some features which worked well for users and which features didn't work well. This contributes to a large but still incomplete body of knowledge about software engineering tools and accounts as some directions tool developers should follow if they are interested in designing and building tools that can efficiently and transparently accommodate to the users' workstyles and usage impulses.

SECTION 6.4 concludes the chapter with a discussion about the limitations of the studies, and the implications the studies present for tool developers.

6.1 Evaluating CanonSketch

Our workstyle model enabled us to discuss and create a new class of design tools that support the increasingly complex tasks involved in modern software development. From our workstyle model we created a new tool, CanonSketch, to support interaction design at multiple levels of detail and using different notations that are particularly effective in different stages of development. The goal of the CanonSketch tool was therefore to support abstract UI design, using the UML semantics, while also leveraging multiple interfaces. The existing version supports rendering for HTML and also MXML. Our long term goal is to support a wide range of device independent markup language such as UsiXML (Vanderdonckt et al., 2004). Despite the long-term usefulness objective of supporting UML based interaction design for multiple platforms, we also wanted to evaluate the usability of our innovative tool against the standard UML-based alternative AMD tools and pencil-and-paper.

We evaluated the CanonSketch tool under two different perspectives: we extracted basic usability engineering results using discount usability engineering (Nielsen, 1993) and we studied the users' behavior according to our workstyle model. To determine if designers would find our tool useful and usable, we conducted two usability tests, aimed at answering questions such as:

1. Can designers use the tool to model the UI at different levels of detail in a *usable* and *useful* way?
2. Does CanonSketch provide the design teams with a standard, non-ambiguous and easy way to *communicate*, thus fostering *collaboration* and sharing of ideas?
3. Does CanonSketch enable fast *comparison* of designs, and does it leverage the *creativity* of designers, thus leading to more innovative prototypes?

In this section, we describe our study design, present the usability results and briefly describe some design findings.

6.1.1 Study Design

We performed the design study in a laboratorial setting with 18 computers running the collaborative version of CanonSketch (in the initial version, without MXML support and

without the code editor, described in the previous chapter). Our subjects were 30 undergraduate students with UML modeling knowledge, OO-programming concepts and practice and basic computer engineering principles.

The main, formal study was conducted in two separate sessions of two hours each. Study A was aimed at evaluating issues in questions 1 and 2 (described in the previous subsection); Study B was aimed at evaluating issues of the question 3.

The design tasks were similar for both studies:

- Study A: In this study, we asked users to design a UML model and an Abstract Prototype for an interface to a weather system for travelers, as (Landay and Myers, 2001) did. This problem is described in Usability Engineering (Nielsen, 1993) and provides a solid benchmark, since possible design errors are well-documented (Nielsen, 1993).
- Study B: In this study, we gave users the task of designing and building creative solutions for a book-selling website UI. We asked them to focus their efforts on designing creative solutions, i.e., to compare and iterate designs towards an inventive prototype.

For more details about the study's design goals, tasks and questionnaires, please refer to Appendix A, CanonSketch's Usability Study Package.

The evaluation for Study A consisted on a between subjects design, where each user acted both as a designer and as a client. In the role of designer, the user had to define/iterate design ideas and communicate/discuss them with the client. In the role of client, he had to understand/discuss the designer's prototype. This study involved 30 users. Half of them used CanonSketch, the other half used Pencil, Paper and Post-it notes.

The evaluation for Study B involved solving the task in groups of two, using CanonSketch only.

6.1.2 Usability Study Results

Based on observing users interact with CanonSketch, answering questions during the tasks, and user discussion after the evaluation, we collected several usability results.

In a post-evaluation questionnaire (see Appendix A), we asked users to rate the ease of change, satisfaction, ease of use, exploration of designs, comparison of designs, ease of

communication and expressive power of their tool on a 7-point Likert scale, with (7) being a highly positive rating and (1) a highly negative rating.

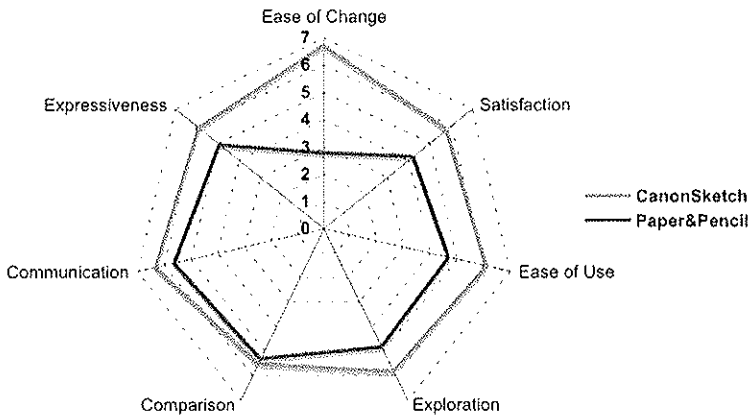


Figure 6.1 Mean values for CanonSketch and Paper & Pencil.

Topic	Mean (Variance)		p-Value
	CanonSketch	Paper & Pencil	
Ease Of Change	6.1 (1.5)	3.7 (1.9)	0.0002
Satisfaction	5.8 (0.7)	4.2 (3.3)	0.004
Ease of Use	6.1 (0.7)	4.7 (3.7)	0.01
Exploration	5.9 (0.6)	4.8 (2.8)	0.03
Comparison	5.5 (1.1)	5.3 (3.1)	0.75
Communication	6.3 (0.7)	5.6 (1.3)	0.01
Expressiveness	5.9 (0.8)	4.9 (3.2)	0.07

Table 6.1 Summary of the Results for Study A.

Figure 6.1 plots the mean values for the obtained results. We can see that CanonSketch easily outperforms the low-tech tools of Paper, Pencil & Post-it notes in all aspects except for the ease of Comparison of designs. Table 6.1 summarizes the statistical results. At a 5 percent significance level, we can see that only Comparison and Expressiveness are not statistically different. In the Ease of Change-related questions, participants using CanonSketch accomplished changes easier (*mean* = 6.1, *var* = 1.5), than those using Paper & Pencil (*mean* = 3.6, *var* = 1.9), $t(28) = 5.1$, $p < .05$. They also felt more satisfied using CanonSketch (*mean* = 5.8, *var* = 0.7) than using P&P (*mean* = 4.2, *var* = 3.3), $t(28) = 3.07$, $p < .05$.

We can see that in all dimensions CanonSketch fares well, with users rating the tool with highly positive values. Compared to using low-tech tools (paper, pencil and post-it notes), CanonSketch is significantly better, except for the Ease of Comparison of designs. This makes sense, since using several sheets of paper and placing them over a table allows observing the "bigger picture" in a natural way. The greatest difference was shown in the ease of change questions: users found CanonSketch to be significantly useful for rapidly changing design elements. This was particularly evident when "clients" added new requirements/suggestions and designers were forced to change their prototypes.

During Study B, participants were asked to rate questions about how well did CanonSketch support their working styles. For this study, we made a similar rating questionnaire as in Study A, but focused our questions on how well did the tool support our workstyle model axes as well as transitions in those axes. Figure 6.2 graphically plots the results and Table 6.2 details the obtained values.

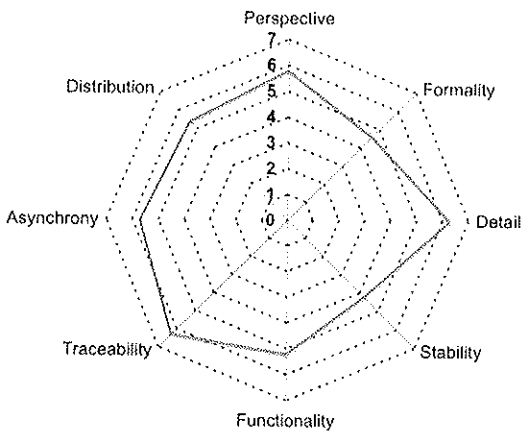


Figure 6.2 The level of workstyle support as evaluated by CanonSketch's users, according to the galactic dimensions.

From the results, we can set out a set of future directions for the CanonSketch tool. The axes that obtained the lowest ratings were Formality (*mean*=4.6) and Stability (*mean*=4.2). We can infer that designers had trouble crafting informal/formal artifacts and also were not fully satisfied with the capability to support fast (or incremental) changes to models.

Functionality also obtained a low value, which is why we added the code editor view in the subsequent version of the tool.

"Galactic" Dimension	Mean (Variance) CanonSketch
Perspective	5.8 (1.2)
Formality	4.6 (3.2)
Detail	6.3 (0.6)
Stability	4.2 (2.1)
Functionality	5.2 (1.1)
Traceability	6.3 (0.7)
Asynchrony	5.7 (1.1)
Distribution	5.4 (2.2)

Table 6.2 Summary of the Results for Study B.

6.1.3 *Design Findings*

Our extensive, 1-semester long study resulted in a series of observations about the users' behavior, which we summarize here in terms of our model's notation, tool-usage and collaboration styles and dimensions.

Notational style issues. One of the interesting observations we made along this study was that users recurred to natural language to describe certain dynamic aspects of the interfaces being prototyped. In addition, some users also showed difficulty on getting a good spatial layout for the abstract user interface, despite the engaging, usable interface of CanonSketch. This suggests that UI tools should provide guidance for the design activities, without establishing constraints on creativity.

Another observation is related to the naming applied by users to their model elements. The act of giving/finding a name for a class, task or screen was sometimes the cause of team misunderstandings and could even lead to bad design choices. Since the method is guided by tasks and essential use cases, users showed a tendency to create too many interaction spaces by not aggregating the ones without sufficient elements to fill up a whole screen.

In spite of a lack of expressiveness for some dynamic aspects we referred above, Constantine's Abstract Prototype notation revealed a much lower learning curve than we expected at the beginning of the study. After only a brief 30 minutes introduction and ex-

amples, users were not confused about it and successfully used the notation to prototype several design solutions.

Tool-usage style issues. We observed gladly that the tools' usability was often appreciated by the subjects. We also observed that the tools' interaction idioms were able to keep users engaged in the development process. Sometimes the users were interested in fully maintaining traceability between all elements in all views and at other times they were not interested at all in maintaining that consistency (they used a single view). This shows the adequacy of the traceability axis in our model.

Collaboration style issues. The collaborative version catalyzed the development process and group discussion. This was surprising, since most of the empirical results on technology-mediated collaboration report degrades in task performance and information sharing (DiMicco et al., 2004; Kiesler and Sproull, 1992; Hollingshead, 1996). We believe much of this is related to the usability of the collaborative version (by using a Messenger-like interface and drag and drop interaction). Users were engaged in both asynchronous/distributed and synchronous/co-located styles of work.

6.2 A Framework for Studying the Designer's Tools and Workstyles

In the survey described in CHAPTER 4 (§ 4.6), we discovered that professional practitioners of interaction design engage into different workstyles throughout their quotidian endeavors. We performed a survey which was distributed to professional interaction designers associations and mailing list, and collected 370 usable responses. This study, which is described in (Campos and Nunes, 2006b) and in Section 4.6 of this dissertation, had two main goals:

- Assess the practical aspects of the workstyle model: in particular if asking questions about workstyles would be feasible and would lead to interesting findings;
- Find interesting patterns of tools' use and/or workstyle transitions among industrial designers.

Results showed that the most frequent and at the same time the most difficult transition was "Moving from business rules, use cases and problem space concepts into final solution design, and back". This is a perspective workstyle transition. Going from problem space to the solution space is clearly the hardest of all workstyle transitions. We gave in CHAPTER 5, as an example of perspective support, the TaskSketch tool, which tries to provide traceability between use cases, the system's conceptual architecture and the UI's initial layout elements.

Another noticeable result was related with current visual interface builders, which were referenced as being used only by 36% of respondents. Because this shows that more research has to be performed into how visual interface builders can better support informal human collaboration, some simple examples were given, in the previous chapter, of how to work towards this goal. The sketch recognition engine in CanonSketch as well as the layer for informally annotating and sketching the final UI were two steps towards this direction. But whether or not the tools themselves remain regarded as being both usable and useful is a question that remains unanswered. In this section, a method for studying *both* the workstyle level support *and* the usability and usefulness of the TaskSketch tool is described. Even if the tool is useful, it doesn't mean that it will actually be *accepted* by the users.

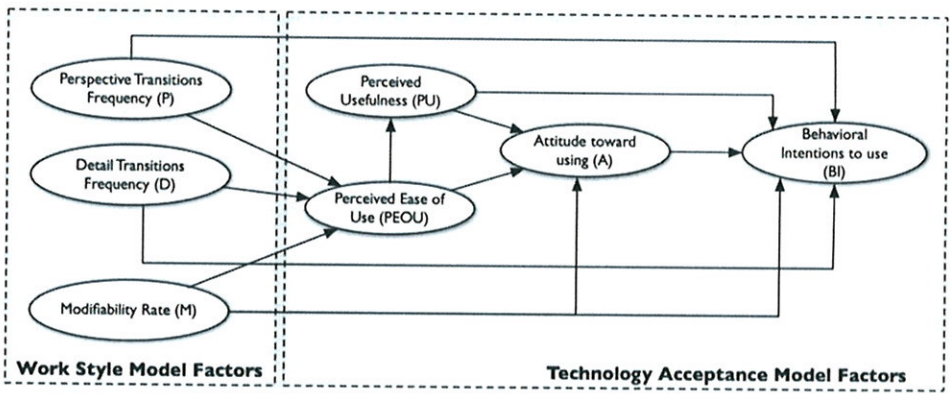


Figure 6.3 Workstyle-related factors in the model (on the left side of the Figure) and the Technology Acceptance Model factors (on the right).

Based on the Technology Acceptance Model (TAM) (Davis, 1989) which was described in § 2.3.1, on current research literature (Abrahão et al., 2005; Morris and Dillon, 2000), on the Workstyle Model for UCD (Campos and Nunes, 2005b) and on our survey's results (Campos and Nunes, 2006b), we designed an experimental framework aimed at studying the interaction designer's tools and workstyles. Figure 6.3 summarizes the constructs in our framework: the constructs related to the Workstyle model are shown on the left side of the Figure (Perspective Transitions' Frequency, Detail Transitions Frequency and Modifiability Rate) and the constructs from the original Technology Acceptance Model are shown on the right side of the Figure (Perceived Usefulness, Perceived Ease of Use, Attitude toward Using and Behavioral intentions to use).

Perception-related variables operationalize the constructs of this framework. Four perception-based variables are measured, just like in the TAM:

- *Perceived Usefulness* (PU) is defined as the degree to which the user believes that using the tool will enhance his or her performance in designing interactive systems;

- *Perceived Ease Of Use* (PEOU) is defined as the degree to which the user believes that using the design tool will be free from effort;

- *Attitude toward using* (A) measures the feelings of favorableness toward using the tool;

- *Behavioral intention to use* (BI) measures the strength of a designer towards using the tool in the near future.

Workstyle-related variables measure some aspects that come from our Workstyle model and from the transitions considered most difficult and frequent by professional interaction designers (according to our survey):

- *Perspective Transitions Frequency* (P) is defined as the rate of transitions from different perspective views, i.e. the frequency of transitioning from problem space concepts (use cases, task flows) to solution space (architecture, abstract prototype) and back;

- *Detail Transitions Frequency* (D) is defined as the rate of "drill-down" or "roll-up" between model elements, i.e. switching from high-detail views of an element to low-detail or the opposite;

- *Modifiability Rate* (M) is the rate of change made to any element of the artifact(s) being designed. This might include changing names, color, size, values or any other property of elements.

Figure 6.4 shows the constructs of our framework as well as all the hypotheses tested. The hypotheses should be interpreted the following way:

- H1 tests whether PEOU significantly influences PU; in other words, it tests whether the users' perception of the tool's usability will contribute to influencing its usefulness;
- H2 tests whether PU influences A;
- H3 tests if PEOU influences A;
- H4 tests if PU influences BI;
- H5 tests if A will influence BI;
- H6 tests if P influenced PU;
- H7 tests if P influenced PEOU;
- H8 tests if D influenced PU;
- H9 tests if D influenced PEOU;
- H10 tests if P influenced A;
- H11 tests if P influenced BI;
- H12 tests if D influenced A;
- H13 tests if D influenced BI;
- H14 tests if M influenced PEOU;
- H15 tests if M influenced PU;

- H16 tests if M influenced A;
- And finally, H17 tests if M influenced BI.

Possible relationships of influence between the workstyle-related constructs were also measured (e.g. by testing if P influences M or if D influences P or M) but we found no significant relationship. Thus, the results presented only try to investigate the actual relation between workstyle-based constructs and technology acceptance constructs (besides testing the original, traditional hypotheses of the TAM model).

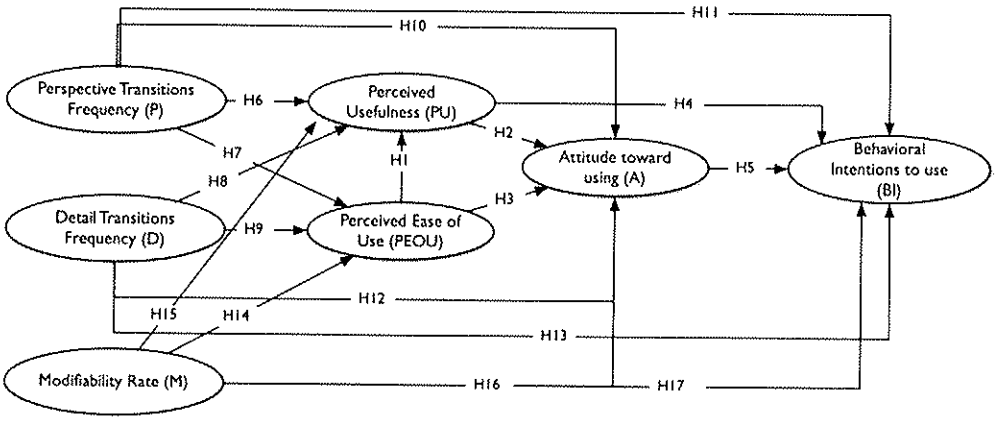


Figure 6.4 The hypotheses and constructs considered in our framework.

As shown in Figure 6.4, our framework is basically an extended version of the TAM; we augmented it by including workstyle objective data that can be automatically obtained from the usage of a given design tool. Both the perspective and detail transitions frequency can be obtained by logging mechanisms. The same happens to the modifiability rate measurement. We augmented TaskSketch in order to produce detailed statistics regarding the following objective variables:

- *Perspective Transitions Frequency*, the number of times (per minute) that a designer switches from a view to another (e.g. use cases view to abstract prototype view); in TaskSketch this is measured by counting the number of times the user switches his focus from one view to another;

- *Detail Transitions Frequency*, the number of times (per minute) that a designer switches from high (low) detail view of the user interface to low (high) detail view; in TaskSketch this is measured by counting the number of times the user switches from the Architectural view (which can be used to define navigation maps of the user interface) to the Abstract Prototype view and back;

- *Modifiability Rate*, the number of times (per minute) that a designer performs changes to any given model element (e.g. changing the text/title of a use case, changing the layout of a prototype, etc.).

A tool (called the Log Analysis Tool) was developed in order to provide support for efficiently and accurately measuring workstyle-related constructs. Figure 6.5 shows a screenshot of the tool. After specifying the log file (which contains a time tag for each relevant action) the tool scans it and gathers workstyle-related data such as the total work time, the percentage of work time spent on each view, the percentage of time per activity (creating new model elements, modifying or searching them),

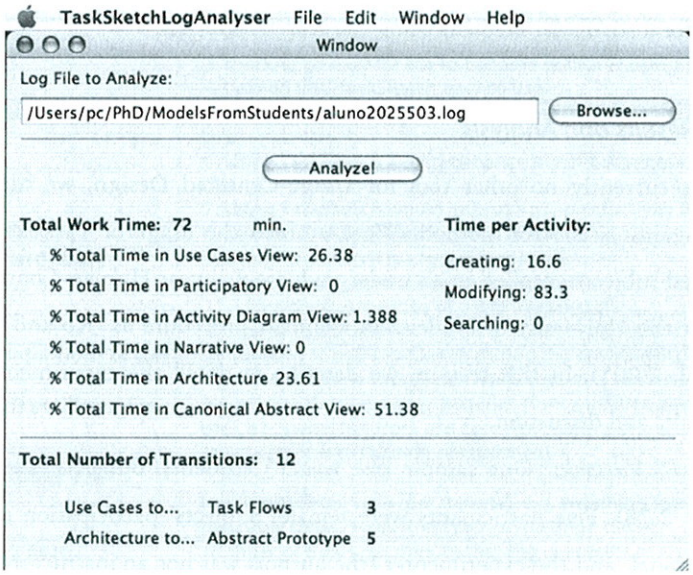


Figure 6.5 The Log Analysis Tool showing workstyle-related data obtained from a log usage file.

We chose to include only the transitions related to Perspective, Detail and Modifiability in our framework and experimental study because of two main reasons: first, according to our survey's results, these are the transitions rated with the highest combination of frequency/cost perceptions (between practitioners). This means they are among the most important ones. And secondly because they are also the simplest transitions to measure. Functionality and Traceability are also easy to measure objectively but are more difficult to implement in a tool. For instance, a tool that supports Functionality transitions should allow designers to create, non-functional, partially-functional or fully-functional UI prototypes. Collaboration-style transitions can only be measured in intrusive ways, e.g. by videotaping users and/or running "think-aloud" protocols (Ericsson and Herbert, 1993; Patton, 2002).

Since we were measuring the rate of modifications to modeling artifacts, we also gathered data regarding the percentage of time users were engaged in *creation* activities (e.g. adding a new use case, drawing a new element, etc.), *modification* activities (e.g. changing the text/title of a use case, changing the layout of a prototype, etc.) and *searching* activities (e.g. looking for a particular element using the search facilities).

6.2.1 Test Results and Analysis

Since there is currently no other tool for Usage-Centered Design, we cannot evaluate TaskSketch against a control method. We examined the usage of TaskSketch in a field study that used subjects enrolled in an undergraduate 3rd year Human-Computer Interaction course at the University of Madeira in a similar procedure as (Ko and Myers, 2005; Abrahão et al., 2005). In this section, we describe in detail the experiment's procedure, variables, results and discussion.

Procedure and Variables. Our sample size was 15 students. Subjects' average age was 22.71 (SD = 2.78). Five participants were female. Subjects' participation on the experiment was optional, and the experimenter (the author) was not an instructor in the course. The experiment took place in a single laboratory equipped with 15 eMacs, and to avoid ceiling effects, we gave no time limit for the execution of the task.

We asked participants to design a Use Case, Task Flow, Conceptual Architecture and an Abstract Prototype for an interface to a weather system for travelers, as Landay and

Myers (2001) did. This problem is described in Usability Engineering (Nielsen, 1993) and provides a solid benchmark, since possible design errors are well-documented (Nielsen, 1993). It is also small enough for a study session and large enough to ensure sufficient coverage of design situations.

After finishing the tasks, subjects were asked to answer a post-experiment survey. The survey included 14 questions, based on the variables of the theoretical model. The items used were formulated through a 7-point Likert scale, where the order of items' presentation was randomized and half the questions negated to avoid monotonous responses and prevent systemic bias in a process very similar to current research such as Abrahão et al. (2005), Morris and Dillon (1997).

We measured quantitative and qualitative variables. The questionnaire we developed uses scales for each subjective variable in the theoretical model presented earlier. The whole set of constructs in this experiment and how they were built and measured are shown in Table 6.3.

PEOU - Perceived Ease of Use	1. Learning to use TaskSketch would be easy to me. 2. It's easy to create models using TaskSketch. 3. It would be easy for me to become skillful at using TaskSketch. 4. I would find TaskSketch easy to use.
PU - Perceived Usefulness	5. Using TaskSketch would improve my performance in designing UI's 6. Using TaskSketch would enhance my effectiveness in designing UI's. 7. Using TaskSketch would improve my productivity in designing UI's. 8. I would find TaskSketch useful in the University.
A - Attitude toward Using	9. Using TaskSketch is a (good/bad) idea. 10. Using TaskSketch is a (wise/foolish) idea. 11. I (like/dislike) the idea of using TaskSketch. 12. Using TaskSketch would be (pleasant/unpleasant).
BI - Behavioral Intentions to Use	13. I intend to use TaskSketch during the remainder of the semester. 14. I intend to use TaskSketch frequently this semester.
P - Perspective Transitions' Frequency	Number of times the user switches from a view to another (per minute), e.g. switching from use cases (problem perspective) view to abstract prototype (solution perspective).
D - Detail Transitions' Frequency	Number of times the user switches from a high (low) detail view to a low (high) detail view (per minute), e.g. switching from navigation map (low detail) to abstract prototype (high detail).
M - Modifiability Rate	Number of times per minute that the user edits or changes any property of any model element, e.g. changing the layout of a prototype, changing a use case description, changing the title of a task.

Table 6.3 Constructs and how they were built.

Validity and Reliability of the Model's Constructs. In order to evaluate the validity of the constructs in our model, we performed an inter-item correlation analysis. We assumed that

all items associated with a particular construct had equal weights. We measured the convergent and discriminant validity proposed by Campbell and Fiske (1959), which was used in several research reports in the SE field (Abrahão et al., 2005).

Convergent validity (CV) assesses if measures of constructs that theoretically should be related to each other are, in fact, observed to be related to each other. It is measured by the average correlation between the indicator and the other indicators that are used to measure the same construct.

Discriminant validity (DV) assesses if measures of constructs that theoretically should not be related to each other are in fact observed to not be related to each other.

The important thing to recognize is that these concepts work together: if one can demonstrate that there is evidence for both convergent and discriminant validity, then by definition, one can demonstrate that there is evidence for construct validity.

	CV	DV	VALID?
PEOU1	0.691	0,503	YES
PEOU2	0.821	0,443	YES
PEOU3	0.698	0,424	YES
PEOU4	0.772	0,457	YES
PU5	0.687	0,440	YES
PU6	0.725	0,529	YES
PU7	0.791	0,478	YES
PU8	0.745	0,491	YES
A9	0.794	0,430	YES
A10	0.795	0,439	YES
A11	0.792	0,644	YES
A12	0.689	0,498	YES
BI13	0.859	0,419	YES
BI14	0.859	0,496	YES

Table 6.4 Correlation between survey items (construct validity analysis).

Table 6.4 shows the correlation results (single inter-item correlation values for every item in the survey was omitted for brevity). For every question in the survey, the convergent validity is always higher than the discriminant validity. This demonstrates the validity for the constructs.

We also performed a reliability analysis on the items of our survey. To ensure reliability of the scales, we calculated Cronbach's alpha for each variable. Cronbach's alpha is the most common form of reliability coefficient. By convention, behavioral studies are considered reliable if the alpha's value is greater than .60 (Nunally, 1978).

Construct	Cronbach's Alpha
Perceived Usefulness	0.902
Perceived Ease of Use	0.898
Attitude toward using	0.899
Behavioral intentions to use	0.937

Table 6.5 Reliability analysis.

The results that prove the reliability of the constructs for subjective evaluation in our survey are shown in Table 6.5. We have obtained high values for the four variables (all alpha values are around 0.9).

Results and Discussion. The descriptive statistics results from the participants' perceptions of use, as well as their attitude and intention to use are shown in Table 6.6.

Descriptive Statistics	PU	PEOU	A	BI
Number of observations	15	15	15	15
Minimum	3.75	2.50	3.50	1.00
Maximum	7.00	7.00	7.00	7.00
Mean	**5.8667**	**5.2667**	**5.6167**	**4.2000**
Standard deviation	1.0892	1.4744	1.2206	1.8400

Table 6.6 Descriptive statistics for PU, PEOU, A and BI.

These results (all averages are well above the 3.5 neutral value in our scale) mean we empirically corroborate that participants find TaskSketch both useful and usable, show a positive attitude towards this tool, and they intend to use TaskSketch in a near future. The ideal method to measure this would be using a control condition, i.e. comparing the data from a group of subjects using another tool against the data from the group using TaskSketch. However, this would be unfair since there is currently no other tool (to our knowledge) specifically aimed at Usage-centered Design. Also, we were interested in obtaining a useful framework for studying workstyle support. Our results can be generalized to the extent that the framework can be applied as a non-obtrusive method for assessing the workstyle-support level of any interaction design tool.

From our log analysis tool, we also extracted the mean frequency of usage time subjects spent on each view. The pie chart in Figure 6.6 shows the distribution of this time. We can see that work at the abstract prototype has the largest usage time, followed by use cases, architecture and finally task flows. We also measured the time spent creating new model elements (mean=28%), modifying model elements (mean=79%) and searching

(mean=1%). All these results make sense from our own modeling tools usage experience, but since there are relatively few quantitative results, this contributes to improve the current set of UI tools' usage data. Also, these results contribute to an increasing body of empirical data about *how* the tools are actually used, and they also assured us of the correct functioning of our logging tool.

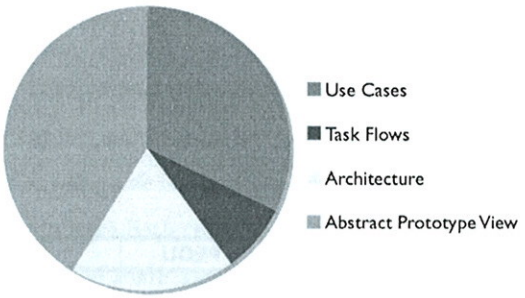


Figure 6.6 Frequency distribution of the several views used by subjects (recorded through automatic logging tools).

Table 6.7 shows the details of our simple linear regression analysis performed to test the hypotheses of our framework. The grey-background rows are the hypothesis taken directly from the TAM model. All the other hypothesis were constructed by us.

Hyp.	Relationship	β	Std. error of β	t	p	R^2	Result
H1	PEOU→PU	0.44	0.164	2.71	0.018	0.362	Supported
H2	PU→A	0.96	0.158	6.13	0.000	0.743	Strongly supported
H3	PEOU→A	0.43	0.196	2.19	0.047	0.270	Supported
H4	PU→BI	0.89	0.397	2.26	0.042	0.282	Supported
H5	A→BI	0.81	0.352	2.32	0.037	0.293	Supported
H6	P→PU	-0.61	0.695	-0.89	0.397	0.202	Not supported
H7	P→PEOU	-2.45	0.530	-4.63	0.001	0.805	Strongly supported
H8	D→PU	-2.16	1.655	-1.31	0.219	0.147	Not supported
H9	D→PEOU	-7.34	1.499	-4.90	0.001	0.706	Strongly supported
H10	P→A	-0.24	0.788	-0.30	0.765	0.243	Not supported
H11	P→BI	-2.65	0.694	-3.82	0.004	0.735	Strongly supported
H12	D→A	-1.83	2.003	-0.91	0.381	0.077	Not supported
H13	D→BI	-8.07	1.768	-4.56	0.001	0.676	Strongly supported
H14	M→PEOU	-0.95	0.333	-2.86	0.019	0.477	Strongly supported
H15	M→PU	-0.46	0.317	-1.45	0.179	0.191	Not supported
H16	M→A	-0.92	0.234	-3.93	0.003	0.633	Strongly supported
H17	M→BI	-0.96	0.380	-2.53	0.032	0.415	Supported

Table 6.7 Regression results. The grey lines show the results for the original TAM model hypotheses (H1-H5). All the other hypotheses were introduced by us.

The regression statistics are the same as the ones described in similar studies, such as Morris and Dillon, (2000). These are the meanings of the regression results' variables:

- β is the regression coefficient (the amount of change in y per unit change in x); in other words, it is the slope of the line which describes the relationship between the independent and dependent variables;

- the standard error of β is the estimated standard deviation of error in estimating y from x;

- t is the statistic for determining whether the relationship is statistically significant;

- p is the statistical significance of the test;

- R^2 tells us the percentage of the variation in y (the dependent variable) that is explained by the scores on x (the independent variable).

We considered a relationship was strongly supported when the level of significance p would be < 0.01 and R^2 greater than 0.4, since in current research results are much more tolerant. Morris and Dillon (1997), for example, consider relationships firmly supported even with R^2 values lower than 0.2.

Figure 6.7 and Figure 6.8 show the results in a visual way. We can conclude that work-style transitions influence significantly (and sometimes very strongly) the perceived ease of use and behavioral intentions of use constructs, while it had no significant effect over the perceived usefulness. The modification rate influenced the attitude toward using (see Figure 6.8).

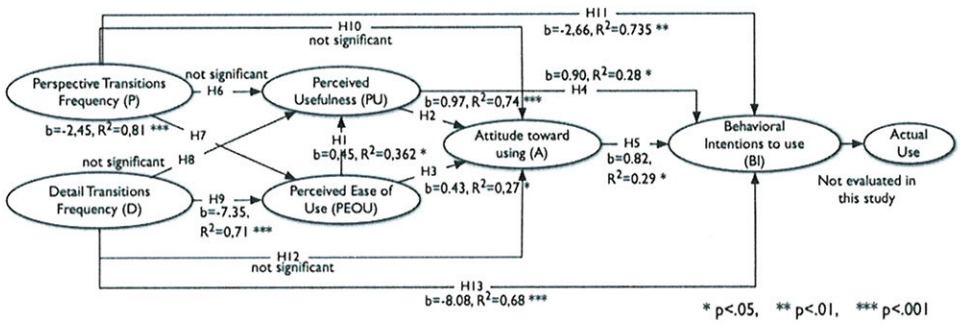


Figure 6.7 Regression results (Modifiability Rate not shown). For each hypothesis tested we present the regression coefficient (b), R-square and the level of significance of the relationship.

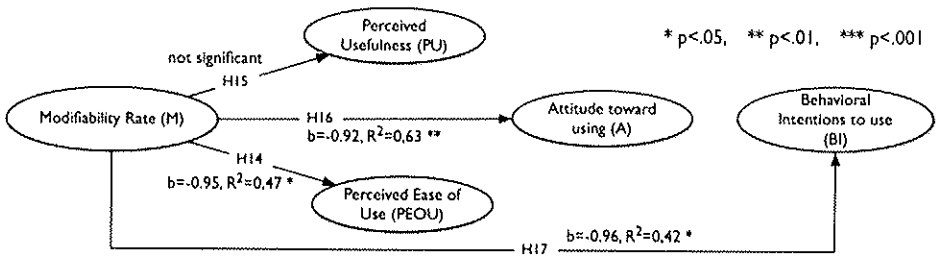


Figure 6.8 Regression results for Modifiability Rate.

In this kind of research activity, it is common to reduce the model to a final, revised and much more compact manner, so that the final model is a significant, descriptive capture of reality, but also the shorter description possible. The final revised theoretical model is presented in Figure 6.9, where we use the stroke width of the hypotheses' arrows to depict the strength of the statistical significant relationships between constructs of our framework (thicker arrows mean a stronger relationship). To summarize, this experiment's results firmly support all the hypotheses derived from TAM. This wasn't surprising, and it does confirm that the study is aligned with many previous related studies, which also validated and confirmed the utility and accuracy of the Technology Acceptance Model.

Regarding the workstyle constructs, these were not found to influence significantly the perceived usefulness of the tool. However, they do play an important role when it comes to perceived ease of use and intentions to using a tool. Detail and Perspective transitions strongly influence Perceived Ease of Use and Behavioral Intentions to use. The Modifiability rate influences not so strongly Perceived Ease of Use and (more significantly) the Attitude toward using, as well as Behavioral Intentions to use.

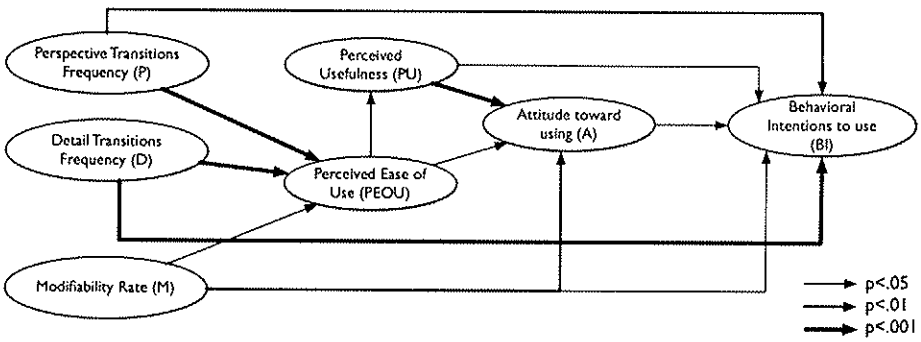


Figure 6.9 The revised framework (after regression analysis).

With this experiment, we have tried to capture the interaction design styles into an empirically-based framework that can be cost-effectively used to study existing design tools as well as to inspire the development of new design tools. We don't claim that our framework completely addresses all the issues related to the current interaction design tools: for example creativity support (Shneiderman, 2002; Jennings et al., 2006) is a recent and hard to measure construct. But we do find our framework is a useful discussion and evaluation instrument. Moreover, it can be used in a non-obtrusive way in order to assess the workstyle-support level of any interaction design tool.

We have augmented the Technology Acceptance Model with quantitative workstyle data from users, which was easy to obtain in a transparent way through the use of logging tools that were also developed and tested throughout this thesis.

It was already known that user perceptions influence software use (Morris and Dillon, 2000). We now know that workstyle transitions can influence the user perceptions for the case of interaction design tools. Therefore, if a better support to workstyle transitions is provided, it might be possible to build better design tools, tools capable of achieving higher adoption levels than current ones. These results suggest that workstyle transitions do have an influence on the tools' perception of usability and usefulness as well as behavioral intentions to use it.

6.3 Principles and Practice

In designing CanonSketch and TaskSketch, many issues were raised both from informal observation of the tools' usage and from formal evaluation studies in laboratorial settings. We will try to summarize a few of the most relevant issues, starting with "what worked well", then describing "what didn't work well".

6.3.1 What worked?

"Semantic" drag and drop. This idea came up as a means to support the Traceability and Perspective dimensions. Designers can drag elements from the activity diagram view (that describes task flows) and drop them in the architecture view, where they are translated to their semantic equivalent. This feature lowered the cognitive load designers were faced when switching views or perspectives and was very appreciated.

Blogs as use case repositories. We wanted to better support collaboration work styles, so we used blogs as common use case repositories fostering offline collaboration (different time, different places). Users can post comments, make changes using web browsers and simply browse the use case narratives, and then designers can choose one or more blog posts and retrieve them into the semantically-sound TaskSketch models.

Brainstorm Collaborative Environment. Also inline with supporting collaboration work styles, we wanted to give users a tool to work at the same time but at different places. The brainstorm collaborative environment (see §5.3.3), where users concurrently post and cluster task cases, classes or UI ideas, proved very attractive to users, probably because of its Messenger-like interface or the dynamics of graphics.

Dynamic Search Box. The dynamic search box that smoothly highlights model elements as the user types proved very useful not only as a search means but also as a way to "filter" desired elements (sharing common properties from the designer's perspective).

6.3.2 What didn't work?

Tool palette as opposed to Drag-and-Drop palette. CanonSketch used a Tool Palette "a la" MS Paint, which users tended to dislike (although appreciating the tooltips). Following

users' suggestions we implemented a drag-and-drop palette ("a la" MS Visual Studio) in the TaskSketch tool. Apparently, users found this approach more usable. One possible explanation for this might be related to the strong bias users have because of being intensive users of tools such as Visual Studio or Visio: both of them strongly-based on model building through drag-and-drop.

Lack of automatic alignment. Clearly these design tools need to improve or adopt automatic alignment mechanisms of some sort. As we have shown, high modifiability rates negatively influence one's attitude toward adopting a design tool, so it is imperative to add this concern when deploying tools for designers. Simply drawing the model elements on the screen, or having a grid layout like MS Visio won't do.

Use Cases manipulation view. Users didn't find comfortable using the "table-like" interface for creating and manipulating the use cases. This is probably related to the process itself: use cases are the central constructs from which everything else is modeled, but they need to be effectively clustered and manipulated.

6.3.3 Principles based on Workstyle Analysis

Based on what we learned during the design and evaluation of both CanonSketch and TaskSketch during a 2-year period, we suggest - and in some cases we try to illustrate - a set of concrete design principles for tool developers.

Explorability. Design tools should make it easy to explore design alternatives. Basic undo-redo mechanisms are a poor man's approach to support this. Designers should not be penalized for trying out other models/solutions and not being able to go back or change ideas. What the Workstyle model also suggests is that tools should invite creativity in early design stages and force focus and discipline later on.

Expressiveness. One of the major areas of weaknesses in current modeling tools is related to its' expressive power. A significant skill required to interaction designers is the ability to express and present their design ideas. Tools should provide helpful mechanisms to achieve this.

Guidance. Depending on the notation being used, the design tool should "just-enough" constrain the actions of the designer. Designers need efficient guidance also on the form of concrete design principles, not canned solutions.

Desirability. Models created with the tools should look engaging and attractive. The user interface should be stylish. Designers are more prone to adopt a design tool based on the visual design of the models created with that tool. A desirable and engaging user interface

In general, we believe that a tool supporting regions of our workstyle model will clearly aid the design and analysis tasks faced by developers of interactive products. Providing support for multiple levels of detail and formality enables designers to start with low-fidelity sketches of products, then gradually add the detail needed to test a prototype. The flexibility achieved by such a tool will most likely contribute to the ease of learning of both tool and method.

6.4 Conclusions

No tool is useful unless it changes its user.
Edgar W. Dijkstra

In this chapter, we have tried to "frame" the designer into a workstyle-based framework that can be effectively and non-obtrusively used to assess a given tool's acceptance level, its usability as qualitatively perceived by the users and its workstyle-support rate. Framing the interaction design styles into an empirically-based model that can be cost-effectively used to study existing design tools can also inspire the development of new design tools. We do not argue that our framework is sound or complete, but we did find it a useful discussion and evaluation instrument.

The Technology Acceptance Model was augmented with quantitative, workstyle-related data from users. This data was easy to obtain in a transparent way that prevents biased responses, which invariably occur when using intrusive evaluation techniques. A tool for processing, analyzing and extracting usage patterns from the log data was also developed and presented.

The obtained results suggest that workstyle transitions can have an influence on the tools' perception of usability and usefulness as well as behavioral intentions to use it. In particular, it was discovered that users who exhibited higher frequencies of workstyle transitions tended to regard the tool as less usable than those users who did not. For example, users that changed model elements at a higher rate were the ones who gave the lower ratings regarding the questionnaire items about the Perceived Ease of Use constructs.

6.4.1 Limitations of the Studies

One of the most important limitations of the CanonSketch study is related to the control method used: comparing CanonSketch against a low-tech tool such as post-its, paper and pencil doesn't take a full picture of CanonSketch's true usability level.

However, if a tool such as ArgoUML had been used to establish a benchmark of comparison, the problem would have remained: ArgoUML, like all other existing tools, is not

specifically designed neither to support workstyle transitions, nor to support the Usage-CD and Wisdom methods. Therefore, the obtained results from such a study would not be fair, since CanonSketch would clearly be in a more advantageous position.

This was one of the reasons why a new framework was designed to better help tool developers assess their tools' acceptance level.

Perhaps the most easily-measured, quantitative measurement of the tools' success is the number of unique, distinct-located downloads which were undertaken by many users: during one year, CanonSketch reported more than 1500 downloads from more than 200 different countries, and TaskSketch reported more than 2000 downloads from more than 300 countries. Taking into account that both tools only work on the Mac platform, a niche market segment with less than 4% share of all world desktop computers, this number becomes even more significant, and is well above our initial expectations.

Perhaps the most significant limitation of the TaskSketch study is related to how well do the results generalize. This is partly due to the small sample size and partly because the population is not sufficiently close to real world designers. However, HCI students are tomorrows' users of the design tools and their perceptions of use map very closely to what a professional designer might think or believe, because their objective is exactly the same. Also, and since we are dealing with perceptions of use and attitudes toward future use, no previous significant experience is required.

6.4.2 *Implications for Tool Developers*

This study carries with it important implications regarding the design of new interaction design tools. Some are based on our own development and evaluation experiences, some are based on the model itself, and some are based on the survey and experiment's concrete results.

Based on our experience and subjects written and oral comments on the tools (both in this study and in a previous one), we believe one of the major areas of weakness of tools is related to the expressive power of tools and the comparison and exploration of alternative designs that a tool can foster. Most of the times it is so much easier to grab a sheet of paper and explain (or explore) a design idea without using a digital tool.

Aesthetics are also important. Most of the positive remarks for our tool are related to how well the models created look both in screen or in print. Workstyle support clearly implicates stylish user interfaces for the supporting tools, an issue that's being more and more debated (Norman, 2004).

If perspective and detail transitions are viewed by professional interaction designers as the most difficult (perspective) and frequent (detail) kind of transitions, and if our results show that these transitions' frequency has negative impacts on the tools' perceptions and intentions of use, then tool designers should find innovative ways to ease those transitions. The same happens with modifiability: the results suggest that the more modifications, the lower the positive feelings regarding the tool. Since we have showed that almost 80% of the time is spent modifying artifacts, effort should be targeted at easing this activity.

If more research is targeted at studying the interaction designer's behavior and workstyles, and if we start designing tools that actually implement these ideas, then the future of interaction design tools will be brighter, because they will designed in order to fit comfortably into the work practices and intentions of their users.

7 Conclusions and Future Work

The software engineering community needs better tools for interaction design, tools that will help ordinary practitioners create extraordinary, high quality user interfaces. Besides the fact that software development companies sometimes don't have the budget for having dedicated interaction designers, it is also a fact of life that software engineers – who are not experts at interaction design – *will* create UIs.

This thesis provides tool engineering aimed at this class of users. What differentiated our design approach from others is the fact that it was specifically aimed at designing in order to support workstyle transitions, in particular some of the most frequent and difficult transitions, as reported by the professionals that answered our survey (Campos and Nunes, 2006b).

The evaluation studies showed that users exhibit a positive attitude towards using both CanonSketch and TaskSketch. The CanonSketch usability study showed that users could use the tool to model a UI at different levels of detail in a usable and useful way. Communication and comparison were also facilitated by CanonSketch. The TaskSketch usability study showed that all values regarding the TAM constructs (Perceived Usefulness, Perceived Ease of Use, Attitude toward using and Behavioral intentions to use) were rated well above the neutral scale, thus corroborating that participants found TaskSketch both useful and usable, and showing a positive attitude towards using TaskSketch in the near future.

Another important finding was related to the influence workstyle transitions has over the tool's perceptions of use. These results suggest that workstyle transitions can influence the perceptions of use for this class of tools, which means it might be possible to design

better design tools if support for these workstyles and workstyle transitions is provided by the tool.

This research has also primed for designing flexible, directly usable and adaptive tools for UI-related activities at both analysis and design level. In parallel, and as part of the design research method (March and Smith, 1995) employed, a model was developed both for describing and evaluating the work-practices support level provided by tools or set of tools.

The workstyle model is useful not only for inspiring design features for tools, but also to clarify the designers' needs and mismatches between activities and tool-support. The model is also effective in assessing a project's effort and stage and can be used to characterize the most typical workflows and transitions. The range of tools designed, developed and presented can support several workstyle transitions, in particular transitions considered by professionals as being frequent and difficult to perform.

We advocate tools that support an effective and transparent adjustment to workstyles, in particular to the workstyles more typically employed by practitioners of UI tasks. And for the first time, there is a fundamental approach that could justify what kind of tool should be used, as well as when and how.

7.1 Limitations

In this section, we will briefly discuss some of the limitations that are present in our current approach. Some of the issues are tool-related, while others concern the evaluation and models developed.

How to handle consistency between models and views is a very important aspect which was not thoroughly studied in the present work. Designing new techniques for keeping models consistent, both at semantic, syntactic and even physically (sticky notes consistent with their digitized version) is a very hard to handle issue.

One limitation of the workstyle model presented in this thesis is that it should be applied to characterize in detail a large portion of existing tools, as well as to characterize the most typical workflow patterns (this was done to some extent). In that way, contrasting the capabilities of existing tools and the actual needs of the designers, and identifying the

areas that lack proper tool support would provide a more solid foundation for the characterization of interaction designers' needs.

Another limitation of our approach is the fact that there is not a simple and clearly defined process of using the CAP notation to specify interfaces for multiple devices. Although CanonSketch can clearly allow multi-platform development (Win, Mac, Palm, Web...), multimodal interfaces are not supported by this tool.

Nevertheless, and even in the absence of model semantics, a tool like CanonSketch has significant value in specifying the architecture of complex interactive systems. Being able to generate HTML and MXML also means the notation is expressive enough to support automatic generation techniques and that it is possible to generate UI's for any platform based on GUI's and Forms like JavaSwing, Palm, Windows or MacOS. After the initial proof of feasibility, we presented a first specification for a UML extension based on the Wisdom notation that was a step towards a full support of Canonical Prototypes in a language that had a major impact on Software Engineering but still remains far from achieving the industrial maturity augured in the 90's, concerning UI modeling.

Regarding the evaluation of the tools, the most important limitation is related with how well do the results generalize. This has a two-fold reason: the size of the sample size (15 for the TaskSketch study and 30 for the CanonSketch session) and the fact that both experiences took place in a controlled laboratorial setting which does not reflect the turbulent environment of small software development companies. More research has to be performed under real world industrial settings, and there is clearly a lack of research in this area, especially regarding the UI-related issues.

Regarding the CanonSketch's evaluation study an alternative evaluation would include comparisons with existing state of the art tools. This is not easy to achieve, however, because currently there is only another tool for supporting part of the Usage-CD process. Comparisons with other tools would therefore be biased because each tool is dedicated to supporting a different process. And one way to view a modeling tool is to see it as a manifestation of a development method, since the tool traditionally delimits your option to following a specific method or technique. Using a tool hence means that the developers perspective is narrowed to what the tool developer finds important.

7.2 Future Work

As with any scientific piece of work, this dissertation has room for many improvements and in this section future possible directions are traced. Some are related with the tools developed, others are connected to the framework and its underlying model.

One of the obvious research directions would be to extend CanonSketch not only with debugging capabilities but also with innovative techniques for code-editing. This would make CanonSketch a truly design and development tool, which would help support the Functionality dimension, thus contributing towards a better integration of software engineers with interaction designers. Another aspect worth investigating is the extent to which the description of UI patterns at different levels of Perspective, Functionality, Detail and Formality is really useful. Only time will tell if designers actually need such a compiled collection of patterns as described in this dissertation. Finally, and still regarding the CanonSketch tool, although its potential regarding the architecture and design of complex, interaction-intensive systems is clear, it could also be possible to study multimodal development capabilities for building non-conventional UIs, using e.g. speech, gestures and sketches (which the CanonSketch tool itself provides).

Regarding the TaskSketch tool, one of the most interesting and promising research direction is related with the participatory, sticky notes-based view. Augmenting TaskSketch with a physical, tangible prototype that would allow a smooth transition between physical artifacts such as index cards and post-it notes towards a digital requirements model would certainly advance the state of the art. Combining the physical affordances of low-tech materials (which foster end users participation in the requirements elicitation and other UCD project's phases) with the advantages provided by digital tools (such as model checking, validation and searching capabilities) is a goal that deserves to be pursuit. This goal is already being made concrete with the definition of a new research project called SMARTCARDS, which could be a step forward regarding a new generation of "media-integrated" UI tools.

As for the workstyle model itself, one possible line of study would be related to analyzing and extracting relevant usage patterns from the log files, thus extending the log analysis tool. One could be able to establish statistically-significant interaction patterns that could be described in the form of rules such as "90% of tool users perform modifications to

artifacts right after creating them". Quantitative approaches of this kind have been successfully proposed for general software engineering (Briand et al., 1992; Peters, 2000; Ernst et al., 2002), but not to UI-specific activities, to our knowledge.

Another improvement to the model would be to discover which style is needed at what step and what transition afterwards. Therefore, according to the development path (e.g., top down, bottom up, or middle out), different transitions might be explored.

An improvement to the evaluation framework suggested in CHAPTER 6 would be to establish a "triangulation method" of evaluation: in other words, design and study an evaluation session that could effectively demonstrate a given tool's level of support for each of the workstyle dimensions. This could be done the following way: notation style-related dimensions (Perspective, Formality and Detail) would be evaluated using a tailored version of Cognitive Dimensions of notations (Green and Petre, 1996). Tool-usage style dimensions (Traceability, Stability and Functionality) would be assessed through Usage-Centered Design's Usability Inspection methods. And finally collaboration-style dimensions would be evaluated using commonly employed CSCW (Computer-Supported Cooperative Work) techniques. Gathering and coupling information from these three sources would allow a "bigger picture" perception of a tool's workstyle support level, not to mention its overall usability.

Appendix A

CanonSketch's Usability Study Package

Designer' Package

Type:	Tool to be used:	Artifact to deliver:
1	CanonSketch	Screenshot of the developed prototype (.pdf) or CanonSketch document file (.csk) + Filled questionnaires
2	A4 Landscape Sheet, Paper, Pencil, Rubber and Post-Its	Sheet with the developed prototype + Filled questionnaires

Design Task

TravelWeather (a fictional system) provides weather information on a periodical basis: 3h, 9h, 15h and 21h, either for the present day as for the next two days. The system should be made available via the Web and should allow the user to:

- Consult weather information regarding any hour/day or regarding a certain region or country;
- Obtain details about temperature, rain, visibility and wind;
- Users are normally tourists heading for long distance destinations and they are interested in knowing the weather conditions in those areas. They first select the region and afterwards the type of weather information they wish to obtain.

TravelWeather wishes to position itself as the leading service for tourists in this market segment.

You should use the Canonical Abstract notation (only) to specify your prototypes.

Design Goals

- Be creative in terms of user interaction, as a way to impress the "client" when he visits you in about 45 minutes.

- The client will only speak to you during 5 minutes, so you should use the attributed tool to build a representation that communicates, in a clear way, your design ideas.
- Don't worry about implementation costs or restrictions in the possible development platforms; be innovative.

Post-Meeting Questionnaire

1. Which of the tools did you use for this task?

❍ Paper & Pencil ❍ CanonSketch

Exploration Capability

2. How would you classify the tool's capability in what regards the ease of exploration of different prototypes? (Circle your answer).

1 2 3 4 5 6 7

(Not Useful) (Neutral) (Very Useful)

Comparison Capability

3. How would you classify the tool's capability in what regards the ease of comparison between different design ideas? Think about the discussions you had with your colleague and with the client.

1 2 3 4 5 6 7

(Not Useful) (Neutral) (Very Useful)

Interaction Ideas Definition Capability

4. How would you classify the tool's capability for express/specify designs? In the case of CanonSketch, developed for Mac OS X, imagine you were a frequent user of this platform.

1 2 3 4 5 6 7

(Not Useful) (Neutral) (Very Useful)

Communication Capability

5. During the meeting with the "client", how was the role of the tool used, in term of communicating to the client your design ideas?

1 2 3 4 5 6 7

(Not Useful) (Neutral) (Very Useful)

Modification Capability

6. During the work session, it might have occurred situations where you and your colleague have decided to significantly changed the prototype. In those situations, how would you classify the modification capabilities of the prototype using this tool?

1 2 3 4 5 6 7

(Not Useful) (Neutral) (Very Useful)

Manipulation/Editing Capability

7. How would you rate the tool's capabilities regarding manipulation and editing facilities, in terms of:

7(a) Ease of placement/removal of components

1 2 3 4 5 6 7

(Difficult) (Neutral) (Easy)

7.(b) Changing the components' relative position

1 2 3 4 5 6 7

(Difficult) (Neutral) (Easy)

7.(c) Changing the components' interactive function and names

1 2 3 4 5 6 7

(Difficult) (Neutral) (Easy)

7.(d) Changing the components' size

1 2 3 4 5 6 7

(Difficult) (Neutral) (Easy)

Ease of Use

8. How would you rate this tool's general ease of use?

1	2	3	4	5	6	7
(Difficult)			(Neutral)			(Easy)

Satisfaction Level

9. How would you rate your satisfaction level after using this tool?

1	2	3	4	5	6	7
(Difficult)			(Neutral)			(Easy)

10. From your experience using the tool, what do you regard as being the tool's strongest aspects?

11. And the weakest?

Client's Package

1. Which of the tools did you use for this task?

❍ Paper & Pencil ❍ CanonSketch

Based on your communication with the designer, enumerate the aspects you consider as the most interesting and positive in the developed prototype. Next, mention the less positive aspects. Note: don't take more than 3 minutes answering this question.

Exploration Capability

2. How would you classify, in terms of your perspective as a client of the designer you visited, the designer tool's capability in what regards the ease of exploration of different prototypes? (Circle your answer).

1 2 3 4 5 6 7

(Not Useful) (Neutral) (Very Useful)

Comparison Capability

3. How would you classify the tool's capability in what regards the ease of comparison between different design ideas? Think about the discussions you had with the designer you visited.

1 2 3 4 5 6 7

(Not Useful) (Neutral) (Very Useful)

Interaction Ideas Definition Capability

4. How would you classify the tool's capability for express/specify the designers' ideas?

1 2 3 4 5 6 7

(Not Useful) (Neutral) (Very Useful)

Communication Capability

5. During the meeting with the designer, how was the role of the tool used, in term of communicating you the design ideas?

1 2 3 4 5 6 7

(Not Useful) (Neutral) (Very Useful)

Modification Capability

6. During the work session, it might have occurred situations where you and your colleague have decided to significantly changed the prototype. In those situations, how would you classify the modification capabilities of the prototype using this tool?

1 2 3 4 5 6 7

(Not Useful) (Neutral) (Very Useful)

Ease of Use

7. How would you rate this tool's general ease of use from your perspective as a client?

1 2 3 4 5 6 7

(Difficult) (Neutral) (Easy)

Satisfaction Level

9. How would you rate your satisfaction level after the designer's presentation, using this tool?

1 2 3 4 5 6 7

(Difficult) (Neutral) (Easy)

Appendix B

A Study on Workstyles and Tools for Interaction Design

Introductory Text

Dear Software Development Professionals,

LabUSE (Laboratory for Usage-Centered Design) of the University of Madeira, is conducting a study on the working styles of interaction designers, information architects and software engineers. Our goal is to characterize transitions in working styles, which is why we ask for your cooperation in answering the following survey. The survey shouldn't take more than 2 minutes, which you can do online through the following URL:

http://www.surveymonkey.com/s.asp?u=342431418638

A few issues regarding this survey:

1. This survey is intended to serve scientific purposes only; we expect to publish results at an international level;

2. Respondents' identities will not be recorded, so privacy is guaranteed;

3. Data will be analyzed in groups and only by the LabUSE team members;

4. If you don't know how to answer a question, or if you have any doubts regarding it's confidentiality, please proceed to the next question.

All results, as soon as possible, will be disseminated to all respondents in the form of e-mail or web page.

Thank you in advance for your attention and help,

LabUSE (Laboratory for Usage-Centered Design) - Univ. of Madeira

Questionnaire

1. What is your primary role in your organization?

- ○ Systems Analyst/Designer
- ○ Programmer
- ○ CIO, Project Manager

- ○ Usability Expert, Interaction Designer
- ○ Other (please specify): .....

2. How many years of professional experience do you have in that role?

- ○ < 2 years
- ○ 3-6 years
- ○ 7-10 years
- ○ > 10 years

3. How many information systems employees does your organization have?

- ○ Less than 5
- ○ 5-14
- ○ 15-49
- ○ 50-100
- ○ More than 100

4. How would you classify your organization software development process?

- ○ "Just do it"
- ○ Waterfall
- ○ Spiral
- ○ Evolutionary development (Prototyping)
- ○ Exploratory development
- ○ Formal methods specifications
- ○ Composition through reusable components
- ○ Other (please specify): .....

5. Which tool(s) do you currently use to perform user interface design? (Check all that apply to any user interface-related activity).

- ☐ Post-it Notes
- ☐ Visual Interface Builders
- ☐ Electronic Sketching Tools (e.g. Denim, SILK)

- ☐ Asynchronous Collaborative Tools (e.g. CVS, e-mail)
- ☐ Analysis and Modeling Tools (e.g. Enterprise Architect, MS Visio)
- ☐ Whiteboards
- ☐ Multimedia authoring tools (e.g. Macromedia Flash, Director)
- ☐ HTML-editing Tools
- ☐ Formal Model-based Tools (e.g. CTT, UI-Pilot, MOBI-D)
- ☐ Synchronous Collaborative Tools (e.g. Video-conferencing, messenger)
- ☐ Paper and Pencil
- ☐ Other (please specify): .....

6. From this point on, you will be given a set of scenarios in terms of workstyles and you will be asked to classify their frequency (how many times you engage and transition in those workstyles) and cost (how difficult you find to perform that transition). Let's start with:

Moving from collaboratively modeling using a whiteboard into a digital version (using e.g. a UML tool).

Frequency:

low			moderate			high
1	2	3	4	5	6	7

Cost:

low			moderate			high
1	2	3	4	5	6	7

7. Moving from high-level descriptions of the user interface (e.g. sitemaps, navigation maps, etc.) to detailed screens of the user interface (concrete widgets, buttons, etc.)

Frequency:

low			moderate			high
1	2	3	4	5	6	7

Cost:

low			moderate			high

1 2 3 4 5 6 7

8. Moving from sketching informal ideas/concepts (using e.g. blackboards or sheets of paper) into formal models (e.g. UML digital models), and back

Frequency:

low moderate high

1 2 3 4 5 6 7

Cost:

low moderate high

1 2 3 4 5 6 7

9. Moving from non-functional prototypes toward fully-functional prototypes

Frequency:

low moderate high

1 2 3 4 5 6 7

Cost:

low moderate high

1 2 3 4 5 6 7

10. Moving from business rules, use cases and problem space concepts into final solution design, and back.

Frequency:

low moderate high

1 2 3 4 5 6 7

Cost:

low moderate high

1 2 3 4 5 6 7

References

A

Abrahão, S., Poels, G. and Pastor, O. (2005). A Functional Size Measurement Method for Object-Oriented Conceptual Schemas: Design and Evaluation Issues. *Journal on Software & Systems Modeling* (SoSyM).

Adobe (2006). http://www.adobe.com

Adobe Labs (2006). http://labs.adobe.com/technologies/actionscript3/

Albrechtsen, H., Pejtersen, A. M. and Cleal, B. (2002). Empirical work analysis of collaborative film indexing. In H. Bruce et al. (eds.), *Emerging Frameworks and Methods: Proceedings of the Fourth International Conference on Conceptions of Library and Information Science*, (pp. 85-108). Greenwood Village, CO: Libraries Unlimited.

Allaire, J. (2002). Macromedia Flash MX - A next-generation rich client. Macromedia whitepaper. http://www.macromedia.com/devnet/flash/whitepapers/richclient.pdf

Alexander, C. et al. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, NY.

Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press, New York, NY.

Aonix (2004). http://www.aonix.com/

Apple (1993). Apple Corporation Newton Personal Digital Assistant.

Apple (2006). Apple Developer Connection, http://developer.apple.com/.

Azevedo, P., Merrick, R. and Roberts, D. (2000). OVID to AUIML - User-Oriented Interface Modeling. In Proceedings of Towards a UML Profile for Interactive System development (TUPIS'00) Workshop, http://math.uma.pt/tupis00, York - UK.

B

Bailey, B. P. (2002). *A Behavior-Sketching Tool for Early Multimedia Design*. Ph.D. Thesis. Minneapolis, MN, Computer Science Department, University of Minnesota: 235.

Bailey, B. P. and J. A. Konstan (2003). Are Informal Tools Better? Comparing DEMAIS, Pencil and Paper, and Authorware for Early Multimedia Design. Proceedings of the ACM Conference on Human Factors in Computing Systems. Ft. Lauderdale, FL: 313-320.

Bailey, B. P., J. A. Konstan, et al. (2001). DEMAIS: Designing Multimedia Applications with Interactive Storyboards. Proceedings of the 9th ACM International Conference on Multimedia. Ottawa, Ontario, Canada: 241-250.

Bastide, R. and Palanque, P. (1990). Petri nets with objects for the design, validation and prototyping of user-driven interfaces. In Proceedings of the INTERACT'90, North-Holland, 1990, pp.625-631.

Beaudouin-Lafon, M. (2004). Designing interaction, not interfaces. In Proceedings of the Working Conference on Advanced Visual interfaces (Gallipoli, Italy, May 25 - 28, 2004). AVI '04. ACM Press, New York, NY, 15-22.

Bellotti, V. and Y. Rogers (1997). From Web Press to Web Pressure. Multimedia Representation and Multimedia Publishing. In Proceedings of Proceedings of Human Factors in Computing Systems: CHI '97. Atlanta, GA, pp. 279-286.

Berti, S. and Paternò, F. (2003). Model-based Design of Speech Interfaces. In Proceedings of DSV-IS 2003, Springer Verlag.

Beyer, H. and Holtzblatt, K. (1998). *Contextual Design: Defining Customer-Centered Systems*. San Francisco: Morgan Kaufmann Publishers, ISBN 1-55860-411-1.

Bias, R.G. (1994). Pluralistic usability walkthrough: coordinated empathies. In J. Nielsen & R.L. Mack (Eds.), *Usability Inspection Methods* (pp. 63-76). New York, NY: Wiley and Sons, Inc.

Bly, S. A. and S. Minneman (1990). Commune: A Shared Drawing Surface. *SIGOIS Bulletin*: 184-192.

Boren, M. T. and Ramey, J. (2000). Thinking aloud: reconciling theory and practice. *IEEE Transactions on Professional Communication* 43 (3), 261–278.

Bouchet, J., Mansoux, B. and Nigay, L. (2005). A Component-Based Approach: ICARE. CHI 2005 Workshop on the Future of User Interface Tools.

Bouillon, L., Vanderdonckt, J. and Chow, K.C. (2004). Flexible Re-engineering of Web Sites. In: Proc. of 8th ACM Int. Conf. on Intelligent User Interfaces IUI'2004 (Funchal, January 13-16, 2004). ACM Press, New York (2004) 132–139.

Briand, L. C., Basili, V. R., and Thomas, W. M. (1992). A Pattern Recognition Approach for Software Engineering Data Analysis. *IEEE Transactions on Software Engineering* 18:(11) (Nov. 1992), 931-942.

Brocklehurst, E. R. (1991). The NPL Electronic Paper Project. *International Journal of Man-Machine Studies* 34(1): 69-95.

Budinsky, F., Finnie, M. and Yu, P. (1996). Automatic Code Generation from Design Patterns. *IBM Systems Journal* 35 (2).

C

Caetano, A., Goulart, N., Fonseca, M., and Jorge, J. (2002). JavaSketchIt: Issues in Sketching the Look of User Interfaces. In Proceedings of the 2002 AAAI Spring Symposium - Sketch Understanding, 9–14, Palo Alto, USA, 2002.

Calvary, G., Coutaz, J., and Thevenin, D. (2001). A Unifying Reference Framework for the Development of Plastic User Interfaces. In: Little, M.R., Nigay, L. (eds.): Proc. of IFIP WG2.7 (13.2) Working Conference EHCI'2001 (Toronto, May 11-13, 2001). Lecture Notes in Computer Science, Vol. 2254. Springer-Verlag, Berlin (2001) 173–192.

Campbell, D. T. and Fiske, D. W. (1959). Convergent and Discriminant Validation by the Multitrait-Multimethod Matrix. *Psychological Bulletin*, 56, 81–105.

Campos, P. and Nunes, N. (2004a). Canonsketch: uma ferramenta para Prototipagem Abstracta e Desenho de Padrões de Interface. In *Actas da 1a. Conferência Nacional em Interacção Pessoa-Máquina*, Lisboa, Portugal, 2004. (In Portuguese).

Campos, P. and Nunes, N. J. (2004b). CanonSketch: a User-Centered Tool for Canonical Abstract Prototyping, In Proceedings of the EHCI/DSV-IS'2004, International Conference on Engineering Human-Computer Interaction / International Workshop on Design, Specification and Verification of Interactive Systems, Hamburg, Germany, Springer-Verlag Lecture Notes in Computer Science.

Campos, P. (2005a). User-Centered CASE Tools for User-Centered Information Systems. In CAiSE 2005, In Proceedings of the 12th Conference on Advanced Information Systems Engineering Doctoral Consortium, 13-14 June 2005, Porto, Portugal.

Campos, P. (2005b). Task-Driven Tools for Requirements Engineering. In Proceedings of the 13th International Requirements Engineering Conference (RE'05), Doctoral Consortium. August 30, Paris, France.

Campos, P. and Nunes, N. (2005a). A UML-Based Tool for Designing User Interfaces. In UML Modeling Languages and Applications: UML 2004 Satellite Activities Lisbon, Portugal, Oct. 11-15, 2004 Revised Selected Papers. Springer-Verlag, 2005.

Campos, P. and Nunes, N. J. (2005b). Galactic Dimensions: A Unifying Workstyle Model for User-Centered Design. In Tenth IFIP TC13 International Conference on Human-Computer Interaction, INTERACT'05 12-16 September 2005, *Lecture Notes in Computer Science*, Volume 3585, pp. 158–169.

Campos, P. and Nunes, N. J. (2005c). A Human-Work Interaction Design Approach by Modeling the User's Work Styles. In Proceeding of the Workshop on Describing Users in Contexts - Perspectives on Human-Work Interaction Design, INTERACT'2005, 12-16 September 2005, Rome, Italy.

Campos, P. and Nunes, N. J. (2006a). Principles and Practice of Work Style Modeling: Sketching Design Tools. In Proceedings of the HWID'06 - Human-Work Interaction Design. IFIP Series, Springer-Verlag, Feb. 2006.

Campos, P. and Nunes N. J. (2006b). Tools of the Trade: the Practitioner's tools and Workstyles. *IEEE Software*, accepted for publication, 2006.

Canyonblue (2004). http://www.canyonblue.com/products.htm

Card, S. K., Moran, T. P., and Newell, A. (1983). *The psychology of human-computer interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Care Technologies, 2006. http://www.care-t.com

Carroll, J. M., ed. (1995). *Scenario Based Design: Envisioning work and technology in system development*. New York: John Wiley and Sons, Inc.

Chesta, C., Fliri, M., Martini, S., Russillo, B., Barbero, C. (2003). First Evaluation of Tools and Methods, CAMELEON Project Document: D 3.4, July 2003.

Clarke, S. (2001). Evaluating a New Programming Language. In Proceedings of Workshop of the Psychology of Programming Interest Group. Bournemouth, UK. pp. 275–89, 2001.

Clemmensen, T., Orngreen, R. and Pejtersen, A. M. (2005). *Describing Users in Contexts - Perspectives on Human-Work Interaction Design*. Workshop Proceedings of Interact'05, Rome, Italy.

Clemmensen, T., Campos, P., Orngreen, R., Wong, W. and Pejtersen, A. M. (2006). *Designing for Human Work*, IFIP TC 13.6 WG Human Work Interaction Design 2006 Proceedings, Madeira Island, Portugal, Springer-Verlag IFIP Series.

Cocreate (2004). Onespace.net: a Web collaboration tool. In http://www.cocreate.com.

Constantine, L. (1994). Modeling Matters. *Software Development* 2 (2), February.

Constantine, L. (1995). Essential modeling: use cases for user interfaces. *ACM Interactions*, 2 (2): 34-46, April 1995.

Constantine, L. and Lockwood, L. (1999). *Software for Use. A Practical Guide to the Models and Methods of Usage-Centered Design*. Addison-Wesley, Reading, MA, pp. 194-195.

Constantine, L. L. (2002). Featured design portfolio. *ACM interactions*, 9 (2), March / April.

Constantine, L. L., and Lockwood, L. A. D. (2002a). Usage-centered engineering for Web applications. *IEEE Software*, 19 (2), March/April, pp 42-50.

Constantine, L. L., and Lockwood, L. A. D. (2002b). Instructive interaction. *User Experience*, 1 (3), Winter.

Constantine, L. L. and Lockwood, L. A. D. (2001). Structure and style in use cases for user interfaces. In M. van Harmelen, Ed., *Object Modeling and User Interface Design*. Boston: Addison Wesley.

Constantine, L. (2003). Canonical Abstract Prototypes for Abstract Visual and Interaction Design. In Jorge, J., Nunes, N. and Falcão e Cunha, J. (eds.), Proceedings of DSV-IS'2003 – 10th International Workshop on Design, Specification and Verification of Interactive Systems, LNCS - Lecture Notes in Computer Science. Springer-Verlag.

Constantine, L. and Campos, P. (2005). CanonSketch and TaskSketch: Innovative Modeling Tools for Usage-Centered Software Design. In Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'05), Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, San Diego, CA, 162–163.

Cooper, A. (1999). *The Inmates Are Running the Asylum: Why High-Tech Products Drive Us Crazy and How to Restore the Sanity*. Sams.

Cooper, A. and Reimann, R. (2003). *About Face 2.0: The Essentials of Interaction Design*. Wiley; 1st edition March 17, 2003.

Coyette, A., Vanderdonckt, J. (2005). A Sketching Tool for Designing Anyuser, Anyplatform, Anywhere User Interfaces. In Proc. of 10th IFIP TC 13 Int. Conf. on Human-Computer Interaction Interact'2005, M.-F. Costabile, F. Paternò (eds.), Lecture Notes in Computer Science, Vol. 3585, Springer-Verlag, Berlin, 2005, pp. 550-564.

D

Damm, C. H., Hansen, K. M. and Thomsen, M. (2000). Tool Support for Cooperative Object-Oriented Design: Gesture Based Modeling on an Electronic Whiteboard. In Proceedings of CHI 2000, The Hague, Netherlands.

Davis, F. D. (1989). Perceived Usefulness, Perceived Ease of Use and User Acceptance of Information Technology. *MIS Quarterly* 13:(3).

Dey, A. (2005). End-User Programming: Empowering Individuals to Take Control of their Environments. CHI 2005 Workshop on the Future of User Interface Tools.

Dillman, D. A. (1999). *Mail and Internet Surveys: The Tailored Design Method*. John Wiley and Sons, New York, 1999.

DiMicco, J. M., Pandolfo, A. and Bender, W. (2004). Influencing Group Participation with a Shared Display. ACM Conference on Computer Supported Cooperative Work (CSCW 2004). Chicago, IL.

Dix, A., Finlay, J., Abowd, G. and Beale, R. (1993). *Human-Computer Interaction*, Prentice Hall, International.

Hix, D. and Hartson, H. R. (1993). *Developing User Interfaces*. John Wiley, New York.

E

Eclipse (2004). http://www.eclipse.org/

Eijk, P.H.J. van and editors (1989). The Formal Description Technique LOTOS.

Embarcadero Technologies (2004). http://www.embarcadero.com/products/describe

Engeström, Y. (1987). *Learning by expanding: An activity-theoretical approach to developmental research*. Helsinki, Finland: Orienta-Konsultit Oy.

Ericsson, K. A., and Herbert A. S. (1993). *Protocol Analysis* (Revised Edition). Overview of Methodology of Protocol Analysis. Massachusetts: MIT press.

Ernst, M. D., Badros, G. J., and Notkin, D. (2002). An Empirical Analysis of C Preprocessor Use. *IEEE Transactions on Software Engineering* 28:(12) (Dec. 2002), 1146-1170.

Everitt, K. M., Klemmer, S., Lee, R. and Landay, J. (2003). Two Worlds Apart: Bridging the Gap Between Physical and Virtual Media for Distributed Design Collaboration. CHI Letters, Human Factors in Computing Systems: CHI2003. 5(1): 553-560.

F

Ferré, X., Juristo, N., Windl, H. and Constantine, L. (2001). Usability Basics for Software Developers. *IEEE Software*, 18(1): 22-29, Jan/Feb, 2001.

Fidel, R. et al. (1999). A visit to the information mall: Web searching behavior of high school students. *Journal of American Society of Information Science*, 50, 24-37.

Fidel, R. and Pejtersen, A.M., (2004). From information behavior research to the design of information systems: The Cognitive Work Analysis framework. *Information Research*, 10.

Foley, J., Kim, W., Kovacevic, S. and Murray, K. (1991). UIDE - An Intelligent User Interface Design Environment. In J. Sullivan and S. Tyler (eds.), *Architectures for Intelligent User Interfaces: Elements and Prototypes*, Addison-Wesley, Reading MA, 1991, pp. 339-384.

G

Gajos, K. and Weld, D. S. (2004). Supple: automatically generating user interfaces. In Proceedings of Intelligent User Interfaces (IUI'99), Island of Madeira, Portugal. ACM SIGCHI, ACM Press, New York.

Gamma et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Gibson, J. J. (1986). *The ecological approach to visual perception*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Ginda, R. (2000). Writing a Mozilla Application with XUL and Javascript. In Proc. of O'Reilly Open Source Software Convention (Monterey, July 19-20, 2000).

Good, M., Spine, T., Whiteside, J. and George, P. (1986). User-Derived Impact Analysis as a Tool for Usability Engineering. In Proceedings of the CHI '86 Human Factors in Computing Systems (1986): 241-246.

Graham, T. C. N., Stewart, H. D. Ryman, A.G, Kopaee, A.R., and Rasouli, R. (1999). A Worldwide-Web Architecture for Collaborative Software Design. *Software Technology and Engineering Practice*, IEEE Press.

Green, T. R. G. and Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7, 131-174.

Griffiths, T., Barclay, P. J., McKirdy, J., Paton, N. W., Gray, P.D., Kennedy, J., Cooper, R., Goble, C. A., West, A. and Smyth, M. (1999). Teallach: A Model-Based User Inter-face Development Environment for Object Databases. In Proc. User Interfaces to Data Intensive Systems (UIDIS99), 5-6th September, Edinburgh, Scotland, pp. 86-96, IEEE Computer Society Publishers, Norman W. Paton and Tony Griffiths (eds.).

H

Hansen, K. M. and Ratzer, A. V. (2002). Tool support for collaborative teaching and learning of object-oriented modeling. In Proceedings of the 7th annual conference on Innovation and Technology in computer science education, pp. 146-150, ACM Press.

Harmelen, M. van (2001). *Object Modeling and User Interface Design*. Addison-Wesley.

Harrison, B. et al. (1998). Squeeze me, Hold me, Tilt Me! An exploration of Manipulative User Interface. Proc. of CHI'98 (1998), 17-24.

Hertzum, M., et al. (2002). An analysis of collaboration in three film archives: a case for collaboratories. In H. Bruce et al. (eds.), *Emerging Frameworks and Methods: Proceedings of the Fourth International Conference on Conceptions of Library and Information Science*, (pp. 69-84). Greenwood Village, CO: Libraries Unlimited.

Hix, D. and Hartson, H. R. (1993). *Developing User Interfaces*. John Wiley, New York.

Hollingshead, A.B. (1996). Information Suppression and Status Persistence in Group Decision Making - the Effects of Communication Media. *Human Computer Research*, 23(2), 193-219.

I

IBM Corp. (2004). Rational Rose Technical Developer. Available from http://www-306.ibm.com/software/rational/.

Ilvari, J. (1996). Why are CASE Tools Not Used? *Communications of the ACM*, 39: 94-103.

ISO (1998). Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs) ISO 9241-11, ISO, Geneva.

Ishii, H., Ullmer, B. (1997). Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms. Proc. of CHI'97, 234-241.

J

Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G. (1992). Object-Oriented Software Engineering: A Use-Case Driven Approach. Reading, MA: Addison-Wesley, 1992.

Jarzabek, S. and Huang, R. (2004). The Case for User-Centered CASE Tools. *Communications of the ACM*, 41 (8): 93-99.

Jennings, P., Giaccardi, E., and Wesolkowska, M. (2006). About face interface: creative engagement in the new media arts and HCI. In CHI '06 Extended Abstracts on Human Factors in Computing Systems (Montréal, Québec, Canada, April 22 - 27, 2006). CHI '06. ACM Press, New York, NY, 1663-1666.

Johnson, J. (2000). *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers (The Morgan Kaufmann Series in Interactive Technologies)*. Morgan Kaufmann.

K

Kaindl, H., Brinkkemper, S. and Bubenko, J. A. Jr. (2002). Requirements engineering and technology transfer. *Requirements Engineering*, 7(3):113-123, Sep. 2002.

Kiesler, S. and Sproull, L. (1992). Group Decision Making and Communication Technology. *Organizational Behavior and Human Decision Processes* 52, 96-123.

Kirakowski, J, and Corbett, M. (1990). *Effective Methodology for the Study of HCI*, North-Holland, Amsterdam.

Kirakowski, J., and Corbett, M. (1993). SUMI: The Software Usability Measurement Inventory. *British Journal of Educational Technology*, Vol. 24, pp. 210-212.

Klemmer, S. (2004). *Tangible User Interface Input: Tools and Techniques*. PhD. Thesis, University of California, Berkeley, Fall 2004.

Klemmer, S. C., Newman, M. N., Farrell, R., Bilezikjian, M. and Landay, J. A. (2001). The Designers Outpost: A Tangible Interface for Collaborative Web Site Design. CHI Letters, The 14th Annual ACM Symposium on User Interface Software and Technology: UIST 2001. 3(2) p. 1-10.

Klemmer, S. R., Li, J., Lin, J. and Landay, J. L. (2004). Papier-Mache: Toolkit Support for Tangible Input. CHI: ACM Conference on Human Factors in Computing Systems, CHI Letters 6(1). pp. 399–406, 2004.

Ko, A. J. and Myers, B. A. (2005). A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems. *Journal of Visual Languages and Computing*, 16:(1-2), 41-84.

Kruchten, P., Ahlqvist, S., and Byland, S. (2001). *User interface design in the Rational Unified Process*. In M. van Harmelen, Ed., Object Modeling and User Interface Design. Boston: Addison Wesley, 2001.

L

Landay, J. and Myers, B. (2001). "Sketching interfaces: Toward more human design." *IEEE Computer*, 34:56–64.

Leont'ev, A. (1978). *Activity, Consciousness and Personality*. Prentice Hall, Englewood Cliffs, NJ.

Lewandowski A., Bourguin G. (2005). Inter-activities management for supporting cooperative software development, Proceedings of the Fourteenth International Conference on Information Systems Development (ISD'2005), Karlstad, Sweden, 15-17 August, 2005, Springer Verlag, 12p.

Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., and Jaquero, V. L. (2004). UsiXML: a Language Supporting Multi-Path Development of User Interfaces. Proc. of 9th IFIP Working Conference on Engineering for Human-Computer Interaction jointly with 11th Int. Workshop on Design, Specification, and Verification of Interactive Systems, EHCI-DSVIS'2004, Lecture Notes in Computer Science, Vol. 3425, Springer-Verlag, Berlin, pp. 200-220.

Long, A. C. (2001). *Quill: A Gesture Design Tool for Pen-based User Interfaces*, Ph.D. Thesis. Berkeley, CA, Computer Science Department, University of California, Berkeley: 307.

M

March, S. Smith, G. (1995). Design and Natural Science Research on Information Technology. *Decision Support Systems*, 15:251-266.

Mathieson, K. (1991). Predicting User Intention: Comparing the Technology Acceptance Model with the Theory of Planned Behavior. *Information Systems Research*, 2:(3), 173-191.

Microsoft (1992). Microsoft Windows for Pen Computing.

Microsoft Corp. (1995). *The Windows interface guidelines for software design*, Microsoft Press, Redmond, WA.

Molina, P. J. (2003). *User Interface Specification: From requirements to code generation*. PhD Thesis, Valencia, Spain.

Molina, P.J., Melia, S., Pastor, O,. (2002). Just-UI: A User Interface Specification Model. In Kolski C., Vanderdonckt J. (Eds.), Computer-Aided Design of User Interfaces III, Kluwer Academic Publishers, Dordrecht, 2002, pp. 63-74.

Molina, P. J., Santiago, M., and Pastor, O. (2002). User Interface Conceptual Design Patterns. In Proceedings of DSV-IS 2002. Rostock, Germany, June 2002: 201-214.

Montero, F. , Jaquero, V. , Vanderdonckt, J. , Gonzalez, P., Lozano, M.D. , Limbourg, Q. (2005). Solving the Mapping Problem in User Interface Design by Seamless Integration in IdealXML. In Proc. of 12th Int. Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS'2005, M. Harrison (ed.), Lecture Notes in Computer Science, Vol. 3941, Springer-Verlag.

Mori, G., Paternò, F. and Santoro, C. (2004). Design and Development of Multi-Device User Interfaces through Multiple Logical Descriptions, *IEEE Transactions on Software Engineering*.

Morris, M. G. and Dillon, A. (1997). How User Perceptions Influence Software Use. *IEEE Software*, 14:(4), 58-55.

Mozilla, (2006). XML in Mozilla, http://developer.mozilla.org/en/docs/XML_in_Mozilla

Mullet, K. and Sano, D. (1995). *Designing Visual Interfaces: Communication Oriented Techniques*. Prentice Hall / SunSoft Press.

Myers, B. and Rosson, M. (1992). Survey on User Interface Programming. In Striking a Balance. In Proceedings of CHI'92, 992: 195-202. ACM Press, Monterey, US.

Myers, B., Hudson, S. and Pausch, R. (2000). Past, Present and Future of User Inter-face Software Tools. *ACM Transactions on Computer-Human Interaction*, 7: 3-28.

N

Nardi, B. A., ed. (1996). *Context and Consciousness*. MIT Press, Cambridge, Massachusetts.

Negroponte, N. (1973). Recent Advances in Sketch Recognition. 1973 National Computer Conference and Exposition. New York, AFIPS Press. 42: 663-675.

Negroponte, N. and Taggart, J. (1971). HUNCH – An Experiment in Sketch Recognition. *Computer Graphics*.

Newman, M. and Landay, J. (2000). Sitemaps, Storyboards, and Specifications: A Sketch of Web Site Design Practice as Manifested Through Artifacts. In Proceedings of the Conference on Designing Interactive Systems (DIS 2000), New York, August 2000.

Newman, M. W., Lin, J., Hong, J. I. and Landay, J. (2003). DENIM: An Informal Web Site Design Tool Inspired by Observations of Practice. *Human-Computer Interaction*, 18: 259-324.

NeXT (2006). NeXT Computer Historical Site, http://next.ex.com/brochure2/10.gif.

Nielsen, J. (1992). Evaluating the thinking aloud technique for use by computer scientists. In Hartson, H. R. and Hix, D. (Eds.), *Advances in Human-Computer Interaction* Vol. 3, Ablex, Norwood, NJ. 75-88.

Nielsen, J. (1993). *Usability Engineering*. Morgan Kaufman, San Francisco, CA.

Nielsen, J. (1999). *Designing Web Usability: The Practice of Simplicity*. New Riders Press.

Nilsson, E. (2002). Combining compound conceptual user interface components with modeling patterns: a promising direction for model-based cross-platform user interface development. In Proceedings of DSV-IS'2003, 10th International Workshop on Design, Specification and Verification of Interactive Systems. Springer-Verlag, 2002.

NoMagic, Inc. (2004). http://www.nomagic.com

Norman, D. A. (1990). *The Design of Everyday Things*. New York, NY: Doubleday.

Nunally, J. (1978). *Psychometric Theory*, 2nd ed., New York, NY, McGraw-Hill.

Nunes, N. J. and Campos, P. (2004). Towards Usable Analysis, Design and Modeling Tools. In Proceedings of the IUI/CADUI'04 Workshop on Making Model-based UI Design Practical: Usable and Open Methods and Tools, Funchal, Portugal.

Nunes, N. J. and Cunha, J. F. (2000). Wisdom: A Software Engineering Method for Small Software Development Companies. *IEEE Software*, 17, 5 (Sep. 2000), 113-119.

Nunes, N. J. and Cunha, J. F. (2001). WISDOM: Whitewater Interactive System Development with Object Models. In Mark van Harmelen (Editor), *Object-oriented User Interface Design*, Addison-Wesley, Object Technology Series, 2001.

Nunes, N. J. (2001). *Object Modeling for User-Centered Development and User Interface Design: the Wisdom Approach*. PhD Thesis, University of Madeira, Funchal, Portugal, April 2001.

Nunes, N. J. (2003). Representing User-Interface Patterns in UML. In Proceedings of OOIS'03 - 9th European Conference on Object-Oriented Information Systems, Geneva, Switzerland, 142-163.

O

Olsen, D. R. and Klemmer, S. R. (2005). The future of user interface design tools. In CHI'05 Extended Abstracts on Human Factors in Computing Systems (Portland, OR, USA, April 02 - 07, 2005). CHI'05. ACM Press, New York, NY, 2134-2135.

P

Paternò, F. and Mezzanotte, M. (1995). Formal Analysis of User and System Interactions in the CERD case study. In Proceedings of the EHCI'95 Conference, pp.213-227, Chapmann & Hall Publisher.

Paternò, F. (2000). *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag, Berlin.

Paternò, F. (2002). CTTE, The ConcurTaskTree Environment.

Paternò, F. (2005). Model-based Tools for Pervasive Usability. *Interacting with Computers*, Elsevier, May 2005, Vol.17, Issue 3, pp. 291-315.

Patrício, L., Cunha, J. F., Fisk, R. and Nunes, N. J. (2004). Customer Experience Requirements for Multi-platform Service Interaction: Bringing Services Marketing to the Elicitation of User Requirements. In Proceedings of the 12th IEEE International Requirements Engineering Conference.

Patton, M. Q. (2002). *Qualitative Research & Evaluation Methods* (3rd ed.), Variety in Quality Inquiry (pp.385). California: Sage Publications.

Pawson, R. R., and Mathews, R. (2002). *Naked Objects*. Chichester, England: Wiley.

Pejtersen, A.M. (1989). The BOOK House: Modeling user needs and search strategies as a basis for system design. Roskilde, Risø National Laboratory. (Risø technical report M-2794).

Pejtersen, A.M. (1992). The Book House. An icon based database system for fiction retrieval in public libraries. In Cronin, B. (Ed.), *The marketing of library and information services* 2. (pp. 572-591).

Peters, J. (2000). *Software Engineering: An Engineering Approach* (Worldwide Series in Computer Science). John Wiley & Sons.

Potts, C. and Catledge, L. (1996). Collaborative Conceptual Design: A Large Software Project Case Study. *Computer-Supported Cooperative Work: The Journal of Collaborative Computing*, 5: 414-445.

Powell, R.A. and Single, H.M. (1996). Focus groups. *International Journal of Quality in Health Care* 8 (5): 499-504.

Preece, J. et al. (1994). *Human-Computer Interaction*, Addison-Wesley Longman, Reading, Mass.

Puerta, A. (1997). A Model-Based Interface Development Environment. *IEEE Software*, 4(14): 41-47.

Puerta, A. and Eisenstein, J. (1999). Towards a General Computational Framework for Model-Based Interface Development Systems. In Proceedings of Intelligent User Interfaces (IUI'99), ACM SIGCHI, ACM Press, New York.

Puerta, A., Micheletti, M., and Mak, A. (2005). The UI pilot: a model-based tool to guide early interface design. In Proceedings of the 10th international Conference on intelligent User interfaces (San Diego, California, USA, January 10 - 13, 2005). IUI'05. ACM Press, New York, NY, 215-222.

R

Rasmussen, J., Pejtersen, A. M. and Goodstein, L. P. (1994). *Cognitive Systems Engineering*. New York: Wiley.

Raskin, J. (2000). *The Humane Interface - New Directions for Designing Interactive Systems*. ACM Press, New York.

Rescher, N. (1973). *The Primacy of Practice*. Oxford, Basil Blackwell, 1973.

Robbins, J. (1999). *Cognitive Support Features for Software Development Tools*. PhD Thesis. University of California, Irvine, USA.

Robbins, J., Hilbert, D. and Redmiles, D. (1997). ARGO: A Design Environment for Evolving Software Architectures. In Proceedings of ICSE'97, pp.600-601. ACM Press, Boston, MA, USA.

Roseman, M. and S. Greenberg (1996). TeamRooms: Network places for collaboration. Conference on Computer Supported Cooperative Work, Boston, MA, ACM.

Rubin, J. (1994). *Handbook of Usability Testing*, John Wiley and Sons, New York, NY.

Rumbaugh, J., Jacobson, I., and Booch, G. (1999). *The Unified Modeling Language reference Manual*. Reading, MA: Addison-Wesley, 1999.

S

Shneiderman, B. (2002). Leonardo's Laptop: Human Needs and the New Computing Technologies. The MIT Press.

Seffah, A. and Kline, R. (2002). Empirical Studies on Software Developers' Experiences. In WESS'02: Eighth IEEE Workshop on Empirical Studies of Software Maintenance. October 2nd, 2002. Montreal, Canada.

Seffah, A. and Metzker, E. (2004). The Obstacles and Myths of Usability and Software Engineering. *Communications of the ACM*, 47(12): 71-76.

Silva, P. P. and Norman, P. (2003). User Interface Modeling in UMLi. *IEEE Software*, 20 (4):62-69.

Singer, J., Lethbridge, T., Vinson, N. and Anquetil, N. (1997). An Examination of Software Engineering Work Practices. CASCON '97, Toronto, October, pp. 209-223.

Snyder, C. (2003). *Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces* (The Morgan Kaufmann Series in Interactive Technologies), Morgan Kaufmann.

Sparks, G. (2004). Enterprise Architect 4.1 User Guide.

Spinellis, D. (2005). The Tools at Hand. *IEEE Software*, 22(1), 10-12, Jan/Feb, 2005.

Standish, (2001). The Standish Group, Extreme CHAOS, Update to the CHAOS Report, available at http://www.standishgroup.com, 2001.

Suchman, L. (1987). *Plans and Situated Actions*, New York: Cambridge U Press.

Suchman, L. (1995). Representations of Work. *Communications of the ACM*, 38:(9), 33-68.

Sumner, T. and M. Stolze (1997). Evolution, not Revolution: Participatory Design in the Toolbelt era. Computers and Design in Context, MIT Press, 1-26.

Sutherland, I. E. (1963). SketchPad: A Man-Machine Graphical Communication System. AFIPS Spring Joint Computer Conference. 23: 329-346.

Szekely, P. et al. (1992). Facilitating the Exploration of Interface Design Alternatives: The Humanoid Model of Interface Design. In Proceedings SIGCHI'92. May 1992, pp. 507-515.

Szekely, P. et al. (1995). Declarative interface models for user interface construction tools: the MASTERMIND approach. In Proc. EHCI'95.

T

Tao, G. and Kokotovic, P. V. (1996). *Adaptive Control of Systems with Actuator and Sensor Nonlinearities*. John Wiley & Sons.

Taylor, S. and Todd, T. (1995). Understanding Information Technology Usage: a Test of Competing Models. *Information Systems Research*, 6:(2), 144-176.

Telelogic Corp. (2004). www.telelogic.com.

Tidwell, J. (2005). *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly Media Inc.

Traeetteberg, H. (2003). Dialog Modeling with Interactors and UML Statecharts - A Hybrid Approach. In Proceedings of DSV-IS 2003 Design, Specification and Verification of Interactive Systems, pages 346-361.

Trætteberg, H., Molina, P. J. and Nunes, N. J., Eds.(2004). *Proceedings of the IUI/CADUI'04 Workshop on Making model-based user interface design practical: usable and open methods and tools*, Funchal, Portugal, 2004.

Turk, M., Robertson, G. Eds., (2000). Perceptual user Interfaces. *Communications of the ACM*, 43, 3, 32-70.

V

Van Welie, M. and Traeetteberg, H. (2000). *Interaction Patterns in User Interface*. In PLoP 2000.

Vanderdonckt, J., Limbourg, Q., Michotte, B., Bouillon, L., Trevisan, D. and Florins, M. (2004). UsiXML: a User Interface Description Language for Specifying Multimodal User Interfaces. In Proc. of W3C Workshop on Multimodal Interaction WMI'2004, Sophia Antipolis, 19-20 July 2004.

Vanderdonckt, J., Berquin, P. (1999). Towards a Very Large Model-Based Approach for User Interface Development. In Paton, N.W., Griffiths, T. (eds.), Proc. of 1st IEEE Int. Workshop on

User Interfaces to Data Intensive Systems UIDIS'99 (Edinburgh, September 5-6, 1999). IEEE Computer Society Press, Los Alamitos, 76–85

Vicente, K.J. (1999). *Cognitive Work Analysis*. Mahwah, NJ : Lawrence.

Vygotsky, L. (1978). *Mind in Society: The development of Higher Psychological Processes*. Harvard University Press.

Visual Basic .NET (2003). http://msdn.microsoft.com/vbasic/

W

Wagner, A. (1990). Prototyping: a day in the life of an interface designer. The art of Human-Computer Interface Design. Addison-Wesley, Reading, MA, pp.79-84.

Walker, M., Takayama, L., and Landay, J. (2002) High-fidelity or low-fidelity, paper or computer medium? Proceedings of the Human Factors and Ergonomics Society 46th Annual Meeting: HFES2002. September 30-October 4, 2002, pp. 661-665.

Wang (1988). Wang Corporation Freestyle tablet computer.

Weinberg, G. (1975). *An Introduction to General Systems Thinking*. Wiley-Interscience.

Welie, M. van and Trætteberg, H. (2000). Interaction Patterns in User Interface. In Proceedings of PLoP 2000.

Wharton, C. et. al. (1994). The cognitive walkthrough method: a practitioner's guide. In J. Nielsen & R. Mack (eds.), *Usability Inspection Methods*, pp. 105-140.

Wiecha, C., Bennett, W., Boies, J. Gould and Greene, S. (1990). ITS: A Tool For Rapidly Developing Interactive Applications. *ACM Transactions on Information Systems* 8(3), July 1990, pp. 204-236.

Wolf, C. G., J. R. Rhyne, et al. (1989). The Paper-Like Interface. Proceedings of the Third International Conference on Human-Computer Interaction: 494-501.

Wong, Y.Y. (1992). Rough and Ready Prototypes: Lessons From Graphic Design. In Proceedings of Human Factors in Computing Systems: CHI '92. Monterey, CA. pp. 83-84, May 3–7, 1992.

Wright, P., Dearden, A. and Fields, R. (2000). Function Allocation: A perspective from studies of work practice. *International Journal of Human Computer Studies*, 52:(2), 335-356.

Wu, J. (2003). Tools for Collaborative Software Design. Technical Report 2003-462, Queen's University, School of Computing, Canada.

Wu, J., Graham, T. C. N. and Smith, P. W. (2003). A Study of Collaboration in Software Design. In Proceedings of the 2003 International Symposium on Empirical Software Engineering (ISESE 2003), IEEE Computer Society, Washington, DC, USA.

Wu, J. and Graham, T. C. N. (2004). The Software Design Board: a Tool supporting Workstyle Transitions in Collaborative Software Design. In *Proceedings of the EHCI / DSV-IS'2004, International Conference on Engineering Human-Computer Interaction / International Workshop on Design, Specification and Verification of Interactive Systems, Hamburg, Germany.*

Z

Zhang, Q. and Eberlein, A. (2002). Deploying good practices in different requirements process models. In Proceedings of the 6th International Conference on Software Engineering and Applications, pp. 76-80, 2002.

Zhang, Q. and Eberlein, A. (2003). Architectural design of an intelligent requirements engineering tool, *Electrical and Computer Engineering*, IEEE CCECE 2003. Volume 2, 4-7 May 2003, pp. 1375-1378.