

A Method for the User-centered and Model-based Development of Interactive Applications

vorgelegt von
Diplom-Informatiker
Sebastian Feuerstack

Von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzende: Prof. Dr. rer. nat. Sabine Glesner
Berichter: Prof. Dr.-Ing. habil. Sahin Albayrak
Berichter: Prof. Dr.-Ing. Sebastian Möller

Tag der wissenschaftlichen Aussprache: 9. Dezember 2008

Berlin 2008

D 83

Zusammenfassung

Die Nutzung von Anwendungen über das Internet ist längst nicht mehr nur über einen Desktop-PC, sondern auch über zahlreiche mobile Endgeräte möglich. Anwendungen und Informationen sind damit theoretisch zu jeder Zeit, von jedem Ort und in jeder denkbaren Situation verfügbar.

Das kontinuierlich wachsende Informationsangebot und die stetig steigende Anzahl von Anwendungen im Internet, die diese Informationen verarbeiten oder die in Zukunft direkt vom Nutzer nach seinen individuellen Bedürfnissen selbst erstellt werden können, benötigen strukturierte, einfach und effizient nutzbare Bedienungsoberflächen. Eine allgegenwärtige Verfügbarkeit dieser Anwendungen, wie es durch die Technologie schon heute möglich ist, erfordert Bedienungsoberflächen, die ihre Darstellung an die Situation des Nutzers anpassen können. Auch die Bedienungsweise sollte konsistent bleiben, wenn der Nutzer seine Bedürfnisse, das Bedienungsgerät, oder die Interaktionsform wechselt.

Bislang ist es nicht gelungen Anwendungen zu entwickeln, die sich selbstständig auf den aktuellen Nutzungskontext anpassen können. Vielmehr werden Anwendungen mehrfach für spezifische Geräte- und Nutzergruppen entwickelt oder die Anpassung der Oberfläche wird ohne Bezug der zugrundeliegenden Semantik der Anwendung vorgenommen. Die Adaption der Oberfläche gelingt demnach bislang entweder nur für Geräte, die während des Entwicklungsprozesses einer Anwendung explizit berücksichtigt wurden, oder geschieht ohne Kenntnis der zugrundeliegenden Anwendung, indem z.B. Bilder grundsätzlich verkleinert oder erweiterte Navigationsmöglichkeiten geschaffen werden.

Diese Arbeit schlägt eine strukturierte Methodik zum Entwurf und der Entwicklung von interaktiven Anwendungen vor, die sich selbstständig auf unterschiedliche Nutzungskontexte adaptieren können. Dabei wird der Schwerpunkt auf der expliziten Beteiligung des Nutzers während des Entwurfs- und Entwicklungsprozesses gelegt. Ziel ist es, die Denkweise des späteren Nutzers zu erfassen und bei der Anwendungsentwicklung zu berücksichtigen.

Die Methodik beschreibt hierzu einen Prozess, in dem das Anwendungsdesign über mehrere Modelle auf verschiedenen Abstraktionsstufen entwickelt wird. Dabei spezifiziert jedes Modell deklarativ eine Sicht auf die Anwendung. Ausgehend von sehr abstrakten Modellen, werden schrittweise konkretere Modelle abgeleitet, die am Ende des Prozesses jeweils eine Oberfläche für eine Gerätegruppe und eine Interaktionsform beschreiben. Durch diese schrittweise Verfeinerung wird die Konsistenz einer Anwendungsoberfläche über mehrere Interaktionsformen und Gerätegruppen hinweg sichergestellt. Die deklarative Semantik der Modelle erlaubt die unmittelbare Interpretation jedes Modells. Somit wird der häufig auftretende Bruch zwischen Designspezifikation und Codierung der Anwendung vermieden. Die vorgeschlagene Methodik wird durch zahlreiche Entwicklungswerkzeuge unterstützt, die alle Phasen des Entwicklungsprozesses

nahtlos abdecken und für jeden Entwicklungsschritt auch die unmittelbare Ableitung eines Prototyps unterstützen. Die Prototypen werden als Beispiel zur unmittelbaren Konfrontation des späteren Nutzers verwendet und erlauben die frühe Berücksichtigung von Verbesserungsvorschlägen im Entwicklungsprozess.

Die in dieser Arbeit vorgeschlagene Methodik wird durch eine Laufzeitumgebung (die „Multi-Access Service Platform“) ergänzt, die mittels Softwareagenten die in der Methodik spezifizierten Modelle direkt ausführen kann. Die Laufzeitumgebung verfügt über Schnittstellen zu den Entwicklungswerkzeugen. Dadurch können Modelle auch zur Laufzeit des Systems mittels der Werkzeuge geändert oder ergänzt werden, um neue Funktionen zu testen oder eine Anwendung um neue Nutzungskontexte zu erweitern.

Es existieren schon jetzt zahlreiche Beschreibungssprachen mit denen eine interaktive Anwendung auf Basis verschiedener Abstraktionsstufen spezifiziert werden kann. Auch wenn sich bislang keine dieser Sprachen durchsetzen konnte, beschränkt sich diese Arbeit darauf, fehlende Modelle zu ergänzen (1), akzeptierte Modellvorschläge zu erweitern (2) und Modellalternativen vorzuschlagen, die bislang noch unberücksichtigt geblieben sind (3).

Zu (1) wird ein Layoutmodell sowie ein Entwicklungswerkzeug vorgeschlagen, mit welchem sich Aussagen spezifizieren lassen, die beschreiben, wie andere Designmodelle bezüglich des Oberflächenlayouts zu interpretieren sind. Zu (2) wird die ConcurTaskTree Notation, welches eine allgemein akzeptierte Beschreibungssprache zur Aufgabenanalyse ist, um die Möglichkeit der direkten Modellinterpretation erweitert und eine explizite Notierung von Domänenkonzepten ergänzt. Zu (3) wird ein alternatives, abstraktes Modell zur geräteunabhängigen Spezifikation von Bedienungsoberflächen entworfen, welches im Gegensatz zu anderen Vorschlägen aus dem Domänenmodell anstelle eines Aufgabenmodells abgeleitet wird.

Die Validierung der Arbeit erfolgt zum einen theoretisch durch Konfrontation der Methodik, der Architektur und der Modellergänzungen mit den Anforderungen aus einer Anforderungsanalyse auf Basis der Evaluation des aktuellen Stands der Technik. Zum anderen wird auf einer praktischen Ebene die Anwendbarkeit der Methodik, Werkzeuge, Architektur und der Modellergänzungen mittels einer Fallstudie, die während eines Forschungsprojektes umgesetzt wurde, detailliert beschrieben.

Abstract

Interactive applications are not bound to traditional desktop computers anymore. With the internet pervading our everyday lives access to information and services is theoretically possible from any device and in every situation. The continuously growing information of the net, and the possibility to build new services based on composing already existing services require well structured and efficient-to-use interfaces. User interfaces need to adapt their presentation based on the actual situation of the user and consistency of the interface needs to be maintained when the user switches the device, modality or the preferences changes.

Nowadays, internet access is available in nearly every situation, but applications fail to adapt to the user's situation. Applications are still developed for a certain device and a specific context-of-use and adaption happens on the user's end-devices without taking into account the applications' semantics. This results in applications that cannot be convenient accessed on a device that the application's designer has not explicitly considered. Currently most of the internet services cannot be comfortably accessed by today's mobile device's browsers that are often only able to scale down images or offer enhanced scrolling capabilities. Further context-of-use adaptations or support for additional modalities have only be realized for very specific applications like for instance in-car navigation systems.

This dissertation research focuses on creating a method for interactive software design that implements a strong user-centric perspective and requires testing steps in all phases of the development process enabling to concentrate on the end-users' requirements and indentifying how the users' way of thinking can be considered as the basic way of controlling the application. The method involves the creation of various models starting by very abstract models and continuously substantiating abstract into more concrete models. All of the models are specified in a declarative manner and can be directly interpreted to form the interactive application at run-time. Therefore all design semantics can be accessed and manipulated to realize context-of-use adaptations at run-time. By the abstract-to-detail modeling approach that includes deriving one model to form the basis of the next, more concrete model, consistency can be achieved if an application needs to be designed to support different platforms and modalities. The method is supported by tools for all steps of the development process to support the designer in specifying and deriving the declarative models. Additionally all tools support the generation of prototypes and examples enabling the developer testing intermediary results during the design process and to consider users' feedback as often and early as possible.

The method is complemented by an associated run-time architecture, the Multi-Access Service Platform that implements a model-agent concept to make all of the design models alive to bridge the actual gap between design-time and run-time of an interactive model-based system. Since the run-time-architecture is able to synchronize changes between

the different models it can be used on the one hand to manipulate a running system in order to prototype and test new features and on the other hand it enables to personalize a running system instance to a certain user's requirements. The architecture considers the tools of the method and enables the designer to deploy changes of the models directly into the running system.

In the past a lot of user interface description languages (UIDL) have been proposed to specify an application on different model abstraction levels. As of now, most of these proposals have not received a broad acceptance in the research community where most research groups implement their own UIDLs. Both, the method and the run-time architecture abstract from a specific UIDL but pay attention to the actual types of abstraction levels that have been identified during the state of the art analysis. Instead of proposing yet another UIDL, the work concentrates on identifying and realizing missing aspects of existing UIDLs (1), in enhancing well accepted approaches (2), and by introducing alternative approaches that have not been proposed so far (3). Regarding (1) the work describes an approach for a layout model that specifies the user interface layout by statements containing interpretations of the other design models and can be designed and tested by using an interactive tool. Specifying a task model has been widely accepted to implement a user-centric development process. Thus, we enhance the ConcurTaskTree notation to support run-time interpretation and explicit domain model annotations to support our work (2). Finally, the work proposes a different way of specifying an abstract user interface model (3) that is based on a derivation based on a domain model instead of using a task model as the initial derivation source.

The validation of this work is done on a theoretical level by confronting the method, the run-time system, and the language extensions to the initial requirements and additionally on a practical level by presenting one of the several case-studies that have been undertaken as part of a research project.

Acknowledgement

First, I wish to acknowledge my principal advisor, Prof. Dr.-Ing. Sahin Albayrak, for his guidance and supervision in helping me cross the finish line. I have been very fortunate to join the DAI-Labor at a time when I needed an advisor and he has helped me through various highs and lows. He has been extremely patient with me during my less productive times, yet been persistent enough in pushing me to complete the research necessary to graduate. In spite of being extremely busy with other students and his other work, there have been few times when I could not just knock on his door and talk to him about my research and other matters.

I am also extremely grateful to Joos-Hendrik Böse, which whom i developed and discussed the initial ideas, which resulted in starting this thesis as well as Marco Blumendorf for his open-minded and constructive discussions and critiques about my work as well as his strong effort joining me to write a lot of research papers and project applications to promote our work on model-based user interface development.

I would also like to thank the other members of my committee: Prof. Dr. rer. nat. Sabine Glesner and Prof. Dr.-Ing. Sebastian Möller. A few other people i would like to thank in the DAI-Labor include the students that have written their diploma thesis or did a project in the topic of model-based user interface development and helped me by testing and implementing various ideas to support this thesis. These are: Nadine Farid, Alexander Nieswand, Grzegorz Lehman, Dirk Roscher, Veit Schwartz, and Serge Bedime.

Thanks also to all the people with whom I shared an office at the DAI-Labor of the Technische Universität Berlin to provide me with an inspiring and enjoyable atmosphere to perform the research during the years to prepare this doctoral dissertation. I also want to thank Deutsche Telekom, and the German Federal Ministry of Economics and Technology for funding the years of my research and thus provide me the opportunity to finish this work.

My parents and family have been a constant source of guidance and support for me throughout my graduate life. Although I have been physically away from them for most of my studies, their encouragement and support has been unwavering.

Berlin, 2008 *Sebastian Feuerstack*

List of publications

This thesis is based on the following research papers:

- [Feu08b]** Sebastian Feuerstack, Marco Blumendorf, Maximilian Kern, Michael Kruppa, Michael Quade, Mathias Runge and, Sahin Albayrak; Automated Usability Evaluation during Model-based Interactive System Development; Accepted for Tamodia 2008
- [Blu08b]** Marco Blumendorf, Grzegorz Lehmann, and Sebastian Feuerstack and Sahin Albayrak; Executable Models for Human-Computer Interaction; DSV-IS 2008, Kingston, Ontario, Canada, 16.-18.7.2008
- [Leh08]** Grzegorz Lehmann, Marco Blumendorf, Sebastian Feuerstack, and Sahin Albayrak; Utilizing Dynamic Executable Models for User Interface Development; DSV-IS 2008, Kingston, Ontario, Canada, 16.-18.7.2008
- [Feu08a]** Sebastian Feuerstack, Marco Blumendorf, Veit Schwartz, Sahin Albayrak; Model-based Layout Generation; ACM Advanced Visual Interfaces Conference 2008; Napoli, Italy
- [Blu08a]** Marco Blumendorf, Sebastian Feuerstack, Sahin Albayrak; Multimodal User Interfaces for Smart Environments: The Multi-Access Service Platform; ACM Advanced Visual Interfaces Conference 2008; Napoli, Italy
- [Blu07b]** Marco Blumendorf, Sebastian Feuerstack, Sahin Albayrak; Multimodal Smart Home User Interfaces; International Workshop on Intelligent User Interfaces for Ambient Assisted Living (IUI4AAL), at International Conference on Intelligent User Interfaces, Spain 2007
- [Feu07]** Sebastian Feuerstack, Marco Blumendorf, Sahin Albayrak; Prototyping of Multimodal Interactions for Smart Environments based on Task Model; Workshop on Model Driven Software Engineering for Ambient Intelligence Applications, European Conference on Ambient Intelligence 2007, Darmstadt, Germany.
- [Blu07a]** Marco Blumendorf, Sebastian Feuerstack, Sahin Albayrak; Multimodal User Interaction in Smart Environments Delivering Distributed User Interfaces; Workshop on Model Driven Software Engineering for Ambient Intelligence Applications, European Conference an Ambient Intelligence 2007, Darmstadt, Germany. 2006
- [Blu06]** Marco Blumendorf, Sebastian Feuerstack, Sahin Albayrak; Event-based Synchronization of Model-Based Multimodal User Interfaces. Second International

Workshop on Model Driven Development of Advanced User Interfaces, MoDELS 2006, Genova, Italy, 1.-5.10.2006

- [Feu06]** Sebastian Feuerstack, Marco Blumendorf, Sahin Albayrak; Bridging the Gap between Model and Design of User Interfaces; Workshop "Modellbasierte Entwicklung von Interaktionsanwendungen", INFORMATIK 2006 Conference, Oct. 2-6, Dresden, Germany
- [Feu05]** Sebastian Feuerstack, Marco Blumendorf, Grzegorz Lehmann, Sahin Albayrak; Seamless Home Services; AmI.d Conference 2006, Sept. 20-22, Sophia-Antipolis, France 2005
- [Rie04]** Andreas Rieger, Richard Cissée, Sebastian Feuerstack, Jens Wohltorf, Sahin Albayrak; An Agent-Based Architecture for Ubiquitous Multimodal User Interfaces; The 2005 International Conference on Active Media Technology, Best paper award, ISSN/ISBN: 0-7803-9036-9, Takamatsu, Kagawa, Japan
- [Feu03]** Sebastian Feuerstack, Axel Hessler, Jan Keiser, and Karsten Bsufka; An Agent-based Framework supporting Rapid Application Development for Telecommunication Applications; Conference on Autonomous Agents and Multi-Agent Systems (AAMAS) 2004; New York;USA
- [Boe03]** Joos-Hendrik Böse and Sebastian Feuerstack; Adaptive User Interfaces for Ubiquitous Access to Agent-based services; Human Agent Interaction Workshop, Agentcities 2003; Barcelona, SPAIN

Contents

1	Introduction	1
1.1	Motivation	1
1.1.1	The Users' Perspective	2
1.1.2	The Developers' Perspective	3
1.2	Thesis	7
1.2.1	Thesis Statement - Aim of this work	7
1.2.2	Validation	7
1.2.3	Scope	7
1.3	Reading Map	9
2	Fundamentals	11
2.1	Gap between Software Engineering and HCI	11
2.2	The model-based user interface development approach	11
2.2.1	Advantages of model-based Development	12
2.2.2	Shortcomings of model-based Development	13
2.3	Terms and definitions	14
2.3.1	Direct manipulation	14
2.3.2	Adaptability and Adaptivity	15
2.3.3	Plasticity and Multi-targeting	15
2.3.4	Multimodal Systems and the Fusion/Fission Problem	16
2.3.5	Context-of-Use	16
2.3.6	Graceful Degradation	17
2.3.7	Platform vs. (End-) Device	17
2.3.8	Designer vs. Developer	17
3	State Of The Art	19
3.1	Methodologies and Tools for Developing User Interfaces	19
3.1.1	Traditional interactive system development	21
3.1.2	Programming by Demonstration / User Interface Prototyping	22
3.1.3	Model-based and task-based approaches	23
3.1.4	The Mapping Problem	34
3.1.5	Conclusions	36
3.2	Architecture Models for Interactive Applications	37
3.2.1	Interaction Model	37
3.2.2	Separation of Concerns	38
3.2.3	Architectures describing Interactive Systems	39

Contents

3.2.4	Conclusions	46
3.3	User Interface Description Languages	46
3.3.1	Concrete Description Languages	47
3.3.2	Model Level Description Languages	49
3.3.3	XIML Meta Model-level Description Language	54
3.4	Conclusions	55
4	The MASP Methodology	57
4.1	Requirements Derivation	57
4.1.1	Observations	58
4.1.2	Shortcomings	59
4.1.3	Methodological Requirements	60
4.2	The MASP User Interface Development Process	62
4.2.1	Analysis Phase	66
4.2.2	Design Phase: Task Modeling	70
4.2.3	Domain modeling	78
4.2.4	Abstract User Interface Modeling	81
4.2.5	Layout modeling	87
4.2.6	Concrete User Interface Modeling	96
4.2.7	Deployment and Implementation phase	104
4.3	Conclusions	105
5	The MASP Software Model for Interactive Systems	106
5.1	Requirements Derivation	106
5.1.1	Architectural Observations	106
5.1.2	Architectural Shortcomings	107
5.1.3	Architectural Requirements	108
5.2	The Multi-Access Service Platform	110
5.2.1	Run-time Interpretation of User Interface Models	111
5.2.2	The Model Agent Concept (MAC)	112
5.2.3	A Model-based Run-time System	117
5.2.4	Layout Constraint Generation at Run-time	119
5.3	Conclusions	120
6	The MASP Abstract Interaction Description Language	122
6.1	Requirements Derivation	123
6.1.1	Language Observations	123
6.1.2	Language Shortcomings	125
6.1.3	Language Requirements	126
6.2	The MASP Abstract Interaction Description Language	127
6.2.1	Task Tree Model	127
6.2.2	Abstract User Interface Model	139
6.2.3	Layout Model	145

Contents

7	The Cooking Assistant Case Study	148
7.1	Analysis Phase: Writing scenarios	149
7.2	Analysis Phase: Identifying tasks and concepts	151
7.3	Design Phase: Modeling the task tree	154
7.4	Domain modeling	156
7.5	Abstract User Interface Modeling	159
7.5.1	Selection activity	160
7.5.2	Context enrichment activity	162
7.6	Layout modeling	166
7.7	Concrete user interface modeling	175
7.8	Service Model	178
7.9	Deployment phase	179
7.10	Conclusions of Case Study	180
8	Conclusions	181
8.1	Context of this Work	181
8.2	Content of this Dissertation	182
8.3	Validation	183
8.3.1	External Validation	183
8.3.2	Internal Validation	184
8.4	Summary of Contributions	191
8.5	Future Work	192
	Bibliography	194
	Annex	215

1 Introduction

Computer systems as we know them are currently changing from single systems running a set of applications to complex networks of connected devices heavily influencing the users' everyday lives. This development leads to new requirements for applications and human-computer interaction. Networks of heterogeneous devices in homes and offices require flexible applications that can adapt to changing environments supporting the user in his daily life - anywhere, on any device and at anytime. This requires interactive systems to be able to switch modalities and devices on demand of the user and to enable user interface distribution combining several devices to control one interactive system.

Addressing various different devices and modalities thus requires the possibility to adapt the user interface to support the specific features of the device's software platform and the provided interaction capabilities. Model-based approaches, practiced for a long time in Human-Computer Interaction seem to be a promising approach to support software developers, developing interactive applications. Model-based approaches aim at modeling the different aspects of the user interface on certain levels of abstraction and offer a declarative way of modeling such multi-platform user-interfaces. With the help of a software engineering process that changes the focus from (manual) implementation to the tool-driven design of models that can be directly executed by a run-time-system, software developers can be supported to address these challenges.

After we give a motivation for our approach in section 1.1 that distinguishes between a users' and a developers' perspective, we introduce this thesis' statements, the validations applied and limiting the scope of this work in section 1.2. Finally in section 1.3 we present the thesis overall structure.

1.1 Motivation

During the recent years information and communication technology are continuously pervading our every day lives. Whereas in early days, only specialists had access to IT technology, nowadays most people have access to technology by the increasing proliferation of fixed and mobile devices [Eisenstein et al., 2001] and demand for ubiquitous access to information. User interfaces that mediate between the human and the machine are required to adapt to the users' and devices' capabilities to be useable for everyone. Realizing user interface adaptation to various kinds of users and devices accounts for approximately 50 percent of total life cycle costs [Myers, 1989]. Costs for developing user interfaces will increase even more when considering an aging society that starts to accept modern IT technology to benefit from mobile communication and internet access but require user interfaces that adapt to their specific demands and disabilities.

The explosion of effort to realize adaptive user interfaces for various platforms and contexts has consequences that need to be considered and handled from two perspectives: the users' as well as the developers' perspective that this thesis focuses on. Both perspectives are described in detail in the following sections.

1.1.1 The Users' Perspective

Users' preferences and interests when using an application are manifold. Depending on each user's individual knowledge about the application domain, the complexity of the user interface needs to be adapted specifically to the user's skills. The actual fixation on visual (graphical-oriented) user interfaces is insufficient for blind or visual impaired people and accessibility mechanisms for them are currently target to peripheral technical software or hardware add-ons such as braille systems that are not directly related to the interactive systems domain but are trying to translate the visual output to voice or haptic feedback based on the users disabilities. Assistive systems in cars or in a manufacture environment require hands-free interactions and therefore other modalities like voice control for instance.

These challenges are summarized by the "Design for All" principle that is synchronous with the "Universal Design" paradigm. The Design for All principle is applied in several disciplines including architecture and product design. Focusing on information technology it relates to the following aspects:

Heterogeneity of users

Whereas in earlier days the users of information and telecommunication technologies were limited to IT professionals, nowadays information technology is available to everyone. But access is still limited to users that have specific skills, including the knowledge about how to control a computer via mouse and keyboard, the way that files and folders are handled and the internet can be accessed. Supporting the user with voice or gesture based interaction access is often a matter of secondary importance. Development and maintenance costs actually rise with each new device or access type supported whereas the earnings shrink as the target group that can be addressed by an additional access technology decreases. Within an aging European society new groups of users emerge that have to be considered by addressing their disabilities, age and knowledge. Up to now it's merely a matter of time to wait until this group becomes huge enough to be specifically considered by future information technology developments on a broad basis. Currently, products that specifically address people with very limited IT knowledge like for instance the Simpleo PC from Fujitsu-Siemens are part of a niche market.

Variety of environments

Mobility becomes the big issue to consider since stationary desktop PC's have been replaced partially by mobile PC's recently and smart phones enable ubiquitous access to IT. Up to now, mobility is more important than easy and comfortable access to services. This has been impressively observed by the unexpected success of the short message

1 Introduction

system (SMS) enabling instant, but very limited communication from everywhere to everyone that owns a cell phone. The future worker is expected to work and collaborate globally and instantly from everywhere and at anytime. Information technology and specifically the user interface has to offer a comfortable control and access even in noisy and busy environments like an airport or a train station or needs to react on direct sunlight during the way to a customer for instance. Additionally globalization and world spanning companies can only be driven by collaborative work between employees of different education and cultures. Supporting their collaborative work via information and telecommunication systems does not only require the localization of the user interface but also its adaption to the individual skills and cultural background of each individual employee.

From the users' perspective this dissertation primarily focuses on the technological diversity, as we think that a proper way to handle and benefit from the technological diversity supports addressing each user individually. The second aspect of universal design, the variety of environments is not explicitly covered in this work, as widening the approach to handle cultural differences and collaborative work is actually handled by another research community and cannot be adequately elaborated as it goes beyond the scope of this thesis.

Nevertheless, we hope that this work will contribute to methodical and architectural aspects to handle the technological heterogeneity as the basic foundation to promote services to adapt to changing environments.

1.1.2 The Developers' Perspective

Changing the view from the users' demands for an interactive application to that of the developers' who are implementing such flexible interactive systems, several challenges can be identified:

Different screen sizes, input capabilities, operating systems and user interface toolkits result in an explosion of effort to realize multi-device applications. User interfaces are often implemented very specific to a certain domain to be generally adaptable. Broadening the access for an interactive application by enhancing a web-based user interface to be accessible additionally on mobile devices does require reducing the amount of information that can be presented on the screen and an adaptation to the specific input capabilities of the device. Pen-based applications for instance allow smaller buttons than touch-screens that require the user's finger to control the user interface. Depending on the user's distance and the screen size of the device the font size and graphics of the user interface have to be adjusted.

Consistency between all platform-specific user interface versions of an interactive application is hard to maintain but necessary to guarantee a seamless interaction across multiple devices [Calvary et al., 2003]. Maintaining multi-device user interfaces results in high development costs since each addition of functionality can require changes to the user interface representation and has to be made for all of the user interface versions.

Often only one view exists to certain functionality. Either the view of the developer's team that realized the application or the view of designers that have created the user

1 Introduction

interface presentation. An evaluation of the application is typically done by carrying out usability tests but is generally considered as too expensive to be applied continuously during user interface development. Additionally there exists no layer of abstraction that can be used to gain feedback during the design phase.

Picture-based mock-ups are used to demonstrate the final presentation of a user interface but are limited to give a non-interactive view to the end-user or voice dialogs are tested using wizard of oz testing experiments but often abstract from the actual state of the technology. Instead of continuous end user testing, semi-structured style guides describing best practices or corporate guidelines are considered during development and end user involvement often is postponed to the beta-testing of the final application.

Missing well structured methods

Different to software engineering processes that have evolved over the last decades, a structured process for designing and implementing multi-platform user interfaces is only applied to a limited extend. Instead of a generally applied method, the designers' creativity and the developers' knowledge about the applications' functional backend are the two main driving forces behind actual user interface developments. Often the process of bringing both views together to form an interactive application is the main area of misunderstandings and conflicts.

For most of the actually available applications a designer has not been included into the process and the developers play the role of the user interface designer. Thus, it's upon the decision of the developer to choose the kind of presentation and its level of verbosity as well as the selection of suitable interface widgets. The selection of suitable widgets is additionally limited by the user interface toolkit that is applied and the developer is bound to programming languages the interface toolkit supports.

Compared to software engineering, in [Limbourg, 2004] several lacks of human computer interaction are listed:

Lack of rigor. The level of rigor in HCI is not the same level as typically used in software engineering [Brown, 1997]. The complexity of the HCI development life cycle has a higher order of complexity than in software engineering [Wegner, 1997]. Currently there are neither automated development processes available that end up with the generation of a user interface, nor can a human alone effectively handle the complexity of developing interactive systems. User interface design requirements are often incomplete or ambiguous and development goals not exactly specified. Further on ill-structured problems miss well-specified criteria for evaluating solutions and have no definite mechanism for applying them.

Lack of systematization. Different to software engineering processes that offer systematization and reproducibility by implementing well-defined and progressive [D'Souza and Wills, 1999] processes, the development life cycle in HCI remains open, ill-defined and highly iterative [Sumner et al., 1997].

Lack of a principle-based approach. Software engineering approaches define system development by progression with one stage after another. In contrast the HCI development cycles typically advance in a more opportunistic way. As soon as an intermediary result is ‘usable enough’, the development proceeds to the next stage [Puerta, 1997].

Lack of explicitness. Experienced designers conduct the development cycle of interactive systems. There are no explicit processes and principles available that capture the knowledge required to properly realize an interactive system [Paternò, 1999, Szekely et al., 1995]. Whereas HCI pattern libraries attained a lot of attention [Tidwell, 2005, Martijn van Welie, 2003] they are mostly limited to specific design issues. Knowledge is currently maintained in the mind of experienced designers.

Missing Development Tools for Multi-Platform User Interfaces

Currently only a few tools that are specifically dedicated to the design of multi-platform user interfaces exist. User Interface design currently happens using GUI builders that are specifically bound to an operating system, a specific programming language and mostly limited to design a graphical user interface for one specific platform. User Interface design is done by sketching pictures using a graphic editor. Beneath the lack of methods for HCI development methods for developing multi-platform user interfaces, only a limited set of tools is available that implement and automate such processes [Florins, 2006]. General knowledge and experience for developing multi-platform user interfaces is only few available. Instead developers are specialized to implement interactive systems for specific target platforms. Each new platform that needs to be specifically addressed, potentially requires a developer that knows about the specific limitations and options of the target platform and ends up in a complete reimplementations of the user interface as there is no shared abstraction between both platforms available that the developer might gain advantage from. Usability guidelines are often not considered appropriately as they are not directly embedded into the development cycle and result in high development costs.

Consistency is the basic principle of user-interface design since the user expects an adaptation to a new platform to rely on his experience of a given version of another platform. If a shared abstraction is missing between the user interfaces for all platforms consistency is very hard to maintain.

Technological diversity

The quickly evolving technological progress has resulted in a huge heterogeneity of devices that people carry along or use stationary in their homes and during work in their company to access services and manage their tasks. Various operating systems and extensive amounts of software are available based on several user interface frameworks. Each of this user interface toolkits has different adaptation capabilities, supports a different set of operation systems and device characteristics. To realize a user interface based on a specific user interface toolkit, the developer needs to be familiar with the programming

language of the toolkit, the characteristics of the supported devices and finally with the methodology describing how the user interface toolkit has to be used to realize a user interface. The same diversity can be observed by the various types of network access technologies that have advanced over the last decade. Access technologies differ in the bandwidth they offer for the interactive application, the level of mobility they support, and the costs of energy and money to utilize them.

Missing Architectures and Frameworks

Nowadays only a very small amount of architectures have been proposed for interactive systems supporting multi-platform interfaces. Actually utilized architectures mainly deal with two aspects: loose coupling and modularization of components.

A loose coupling of the user interface components and the application's core functionality has been only partially reached. So far only patterns have been proposed that focus on a microscopic, implementation level of source code components. On this level patterns proposed the separation of the application's backend from its actual view to the user like described by the popular MVC pattern for instance. Currently an application is often tied to one operating system and thus heavily depends on the underlying interface technology and its application scope is limited to the included user interfaces.

Component modularization has been promoted to handle the complex eventing mechanisms of graphical user interfaces where a lot of elements presented simultaneously are related in some way. As graphical user interfaces typically offer various alternative ways to control an interactive system, component modularization and organization helps to keep the eventing between components in structured systems. A popular paradigm is the Presentation Abstraction Control approach that we will discuss in detail when we present our approach for an architecture that provides multi-platform user interfaces.

Beneath component modularization and loose coupling, the architecture of an interactive system has to be flexible for changing requirements [Luo, 1994].

Those can be targeted by redefinitions of the organizational structure of the interactive system and by relocating tasks that are supported by the system. Further on the system might need to be adapted to reflect a dynamic business environment that allows tasks to be transferred from one device to another.

Especially large interactive systems require a framework to support implementing changes on different levels of abstractions providing the developer with a comprehensive overview of the system to be able to estimate the consequences arising from the changes considered. This manipulations on a certain level of abstraction need to be propagated back to the other levels in an automated or semi-automated way that always ensures consistency along all levels of abstraction considered by the architecture.

1.2 Thesis

1.2.1 Thesis Statement - Aim of this work

This work focuses on a model-based approach for developing device-independent user interfaces formalizing the development of user interfaces for different end-devices to address the shortcomings outlined in the previous chapter. Therefore this dissertation provides:

1. a user interface development methodology, which extends common software engineering approaches to be applicable for the development of device-independent user interfaces. The methodology introduces a concept called “*Design by Example*” enabling short user interface development iterations and is open for instant testing in all phases of the development process.
2. a model-based architecture for user interface provision allowing the explicit definition of user interface models on different levels of abstraction to reflect the development steps of the methodology. The architecture extends earlier architectures for interactive systems by bridging the gap between design and run-time of a system. Therefore the architecture is awakening the design models alive and making them available for manipulations at run-time. Further on, it is open to support various user interface description models, as it offers a meta-model layer allowing to flexibly introduce new models based on the actual requirements of the interactive system.
3. three extensions to already existing user interface description languages to describe an interactive application. First, we extend a task analysis and design notation to enable direct interpretation at run-time and rapid prototyping to test new forms of interactions. Second, we propose an abstract user interface model that is strictly derived from a domain model to preserve its semantics. Third, we propose a model-based layouting approach enabling a consistent layout derivation for various graphical end-devices and a flexible adaptation to new end-devices at run-time that have been unknown at design-time. All extensions are embedded into the proposed development process and benefit from the model-based run-time environment.

1.2.2 Validation

The validity of the approach described in this thesis is tested on a theoretical level by confronting the methodology, the model-based run-time architecture, and the language extensions to the requirements identified after the state of the art analysis of already existing approaches. Additionally a practical validation is provided by illustrating how the methodology and the run-time architecture have been used by presenting a case study that has been undertaken as part of a research project at the DAI-Labor.

1.2.3 Scope

The scope of this work is located in the discipline of *Engineering for Human-Computer Interaction* (EHCI), which is a crossroad of two disciplines: *Human Computer Interaction*

1 Introduction

(HCI) and *Software Engineering (SE)*. HCI can be defined as “a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them.” [Hewett et al., 1992].

This work targets two group of users, end-users and software developers. End-users benefit by using the run-time environment allowing user-interface adaptation at run-time and therefore having a bigger chance to get the user-interface that matches best their preferences regarding layout, language and end-device. Disabled people can benefit from this run-time-environment as well, as the generated user-interfaces can be adapted to their specific requirements much easier as without a model-based approach.

Software developers are the main audience for the approach presented in this thesis. The model-based approach and the methodology bridge the gap to commonly used software engineering methods that are well known to software developers.

In the Human-Computer Interaction discipline there is a lot of research ongoing about the usability of user-interfaces. Usability of user interfaces is not the main focus of this dissertation, which focuses more on the technical challenges of device-independent user interface generation. But since model-based approaches systematically derive user interfaces for various platforms based on a well structured model to model derivation, this allows to make sure that consistency of the user interface can be maintained for all adaptations of the user interface. The proposed user interface development method concentrates on the human as the main source of feedback during the design phase.

It is a common understanding that “*user interface design is an art* “ [Laurel, 1990] and we share this viewpoint. During the research for this work it had become clear, that an automated process for user-interface design has to be supported by a methodology and coupled to a run-time environment to reduce development time and to raise product attractiveness by generating flexible user interfaces for a broad range of devices in one process. But up to now the proposed mechanisms are not able to automatically generate “usable” or high-attractive user-interfaces.

Over the last decades a lot of user interface description languages have been proposed to enable the generation of multi-platform user interfaces. We discuss these approaches in detail in the state of the art chapter and present specialized languages to support certain design aspects, as well as comprehensive UIDLs. Thus, it is not the focus of this work to reinvent the wheel but instead to concentrate on gaps that occurred for realizing the requirements. In the scope of this thesis we concentrate of three UIDL extensions. First, an adaptive and model-based layouting of graphical user interfaces and second, a task notation extension supports to be interpreted at run-time to enable rapid prototyping of new platforms and modalities. Finally, we propose an abstract and therefore platform independent user interface model that is derived based on the domain model semantics. Multimodal user interfaces are only addressed for prototyping scenarios by this work. Especially user interfaces that support the simultaneous usage of several modalities at once require extensive fusion and fission mechanisms and are beyond the scope of this work.

1.3 Reading Map

This work is organized into eight chapters. After the introduction which includes presenting the motivations for this work, the thesis statements and a section about the scope of this work, we introduce in chapter 2 the fundamentals that have to be known and reflect the actual relevant ongoing discussions in the model-based user interface development research community. Therefore we shortly give an overview about the general motivation of model-based development of user interfaces and describe the actual advances and shortcomings of following such an approach. Thereafter we introduce the main terms and definitions that support our work.

In chapter 3 we describe the significant pieces of existing work in the state of the art section. The state of the art analysis has been undertaken by discussing related work regarding three aspects for model-based user interface development. First the relevant work regarding methods and tools, second actual architectures to specify interactive applications are discussed, and finally an overview about relevant user interface description languages is given.

Thereafter three chapters present the main contributions of this dissertation: the method (chapter 4), the run-time architecture (chapter 5) and the UIDL extensions (chapter 6). For each chapter the overall structure remains the same: After an introduction we derive the requirements based on the state of the art analysis of chapter 3 by applying three systematic steps: First, the *observations* are presented, which are synthetic and descriptive assessments (as opposed to normative assessments) and are made regarding to the properties of the surveyed models, processes and languages that are part of the whole user interface development process. From the observations *shortcomings* are outlined. Shortcomings are normative assessments made regarding to properties of the surveyed models, processes and languages. The identification of normative assessments is done by positioning the state of the art with respect to ideal properties. Finally the *requirements* are identified that are proposed as a solution to the shortcomings listed before. After the requirements have been identified, we detail our approach to tackle the requirements in the main section of each chapter and present validations that we did to test and evaluate our work. We close by summarizing each chapter in the conclusions section. Realizing this structure the three main chapters deal with the following aspects:

Chapter 4 introduces the reference methodology that is used to bridge the gap between software engineering and user-interface development. It starts by describing the user interface development process and presents the “Design by example approach” we propose for a more efficient development of device-independent user interfaces. All parts of this user-interface development methodology are tool-supported. For some phases of the development process already existing tools can be used, whereas others are implemented for this work to evaluate the methodology within several research projects.

Chapter 5 introduces the Model-Agent Concept enabling to realize a modular run-time environment for model-based interactive systems. After the actual supported models have been presented, two aspects of the run-time environment are described in more detail, as they are topic of extensive research in the research community for model-based user interface development: The meta-modeling allowing to introduce new models to the

1 Introduction

architecture and the model-mapping required to combine the user-interface models.

In chapter 6 we present three language extensions that we require to enable a task based prototyping in our run-time environment, an abstract user interface model that is derived from a domain model, and a layouting model. These extensions are embedded into both, our method as well as our run-time architecture.

Chapter 7 presents a case-study in order to validate both, the user-interface development process and the model-based run-time environment on a practical level. Referring to the results of a research project that have been carried out during the work on this dissertation the whole development cycle regarding the user-interface development is demonstrated and evaluated as a case study.

Chapter 8 concludes by discussing the appropriateness of the solution proposed in this dissertation by validating the results against the requirements that have been formulated in the previous chapters. The contributions are summarized and future works are proposed.

2 Fundamentals

Before we start inspecting the related work in the next chapter, we shortly introduce the most important terms and concepts in model-based interactive system development. Therefore the next section describes the actual gap between software engineering and HCI that can be seen as the initial idea behind model-based ui development. In section 2.2, we present the actual advantages of model-based approaches and confront them with the discussion about shortcomings. The chapter ends by introducing the basic terms and definitions relevant for this work in section 2.3.

2.1 Gap between Software Engineering and HCI

Whereas the Human Computer Interaction (HCI) “discipline (is) concerned with the design, evaluation and implementation of interactive computer systems for human use and with the study of major phenomena surrounding them” [Hewett et al., 1992], software engineering can be defined as “the application of a systematic, disciplined, quantifiable approach to development, operation, and maintenance of software; that is, the application of engineering to software” [IEEE Computer Society, 1990]. The quality of interactive system development mainly depends on the skills of the developers and can be often measured after most of the development has taken place by applying user tests. Gained experience is often stored in the developers’ minds and is not considered for updating the applied method. This is a very different approach than in sciences like for instance mathematics or physics as they apply a decent level of rigor by following a well defined method resulting in a more structured, predictable, and controllable development process [Limbourg, 2004].

Model-based approaches have been developed over the last years introducing well defined development processes to bridge the actual gap between HCI and software engineering. The next section introduces the basic ideas of a model-based user interface development approach and presents the pros’ and cons’ when following such an approach. Thereafter, in subsection 2.3 some of the major terms and definitions in the model-based user interface development area are briefly presented and related to the topics of the thesis.

2.2 The model-based user interface development approach

Abstraction is a very natural way that people usually like to do to manage too complex entities. To manage complex problems people usually start to discover the main aspects that should be taken into account as well as their relations. An examples for identifying

models in our daily practice is for instance our every day planning. After waking up we typically start the new day by thinking about the main activities that we would like to perform [Paternò, 2005].

The development of interactive systems is a complex problem that is continuously growing with the evolving technology enabling to consider the context-of-use during interaction or multi-modal access to an application. Model-based approaches, practiced since the late 1980's in Human-Computer Interaction seem to be a promising approach to support software developers, developing interactive applications. They aim at modeling the different aspects of the user interface on certain levels of abstraction and offer a declarative way of modeling such multi-platform user-interfaces. Model-based user interface development is concerned with two basic topics:

1. The identification of suitable sets of model abstractions and their relations for analyzing, designing and evaluating interactive systems
2. The specification of software engineering processes that change the focus from (manual) implementation to the tool-driven design of models

Over the last years several pros and cons for following a model-based development approach for user interface design have been identified and intensively discussed. The following section introduces the major advantages of following a model-based development approach that motivated us to further explore the systematical development of user interfaces by relying on models. Thereafter actual shortcomings are presented that this work deals with. Initial ideas how to handle and overcome these shortcomings are briefly introduced and are addressed in the subsequent chapters.

2.2.1 Advantages of model-based Development

Advantages of model-based user interface development approach can be identified for three topics: methodology, re-usability, and consistency [Puerta, 1997]:

Advantages in terms of methodology Model-based approaches are driven by specifications that are subsequently derived by a predefined process. Starting the development cycle with a specification is a widely accepted software engineering principle as [Ghezzi et al., 1991] notes. User-centred and user interface-centred development life cycles are supported. They let designers work with tasks, users, and domain concepts instead of thinking in engineering terms [Szekely et al., 1995]. These models encourage to think more about artefacts that should be realized and force the designers to explicitly represent the rationale of design decisions [Szekely et al., 1995].

Relying on declarative models is a common representation that design tools can reason about to criticize designs and to detect questionable features [Braudes, 1990, Byrne et al., 1994]. Declarative models enable realizing automated advisers that can support the designer to refine the designs. Further on user interface construction tools can be based on declarative models that enable automated creation of portions of the

user interface. During run-time a declarative representation by models can be used to generate context-sensitive help to assist the user like proposed in [Sukaviriya, 1988].

An interactive system specification by using models enable executing the system before all details of the user interface have been designed to enable early experiments with designs by an iterative development process before considerable coding effort has been spent [Szekely et al., 1995].

Advantages in terms of re-usability For multi-platform development of user interfaces and user interface support for context dependent adaptations model-based tools can provide the fundament for an efficient development life cycle that offers an automatic portability across different devices. Furthermore the complete description of the whole interface in a declarative form allows reusing the most interesting components.

Advantages in terms of consistency Consistency is the big issue that needs to be guaranteed between the user interfaces that are generated for different target platforms. Model-based approaches allow some form of consistency between phases of the development cycle and the final product.

2.2.2 Shortcomings of model-based Development

Several shortcomings have been identified so far that need to be tackled down. Commonly cited are [Puerta, 1996, Szekely, 1996, Myers et al., 2000]:

High threshold The high threshold of model-based approaches is one of the big issues that need to be solved to get a broader acceptance. Up to now, developers need to learn a new language in order to express the user interface specification. Thus, model-based approaches require models to be specified in special modeling languages and therefore require a form of programming that is not suitable for many interface developers or designers.

Design tools that enable visual programming by abstracting from a certain user interface language and are integrated in widely deployed development environments are a solution to lower the threshold for model-based approaches.

Unpredictability Each abstraction by a certain model requires the designer to understand and think in the same abstractions of the model that is utilized. The higher the abstraction that the model offers compared to the final user interface, the harder it gets for the designer to understand how the model specifications are connected to the final user interface.

[Florins, 2006] proposes to rely on explicit transformation rules with a tool-based preview to reduce the unpredictability of model-based approaches. [Limbourg, 2004] applied such a graph-based transformational approach, but the pure amount of transformations to map between the various models of abstraction figured out hard to overview and

maintain. Additionally, the wide range of user interface element concepts of the various platforms that need to be covered by the model-transformations is hard to handle and selecting the appropriate transformation between several alternatives (often there are several ways to realize a user interface) emphasizes as a difficult and challenging task.

Propagation of modifications Supporting several models of abstraction for the design and specification of an interactive system includes allowing changes to each of these models later on. These changes need to be propagated to the other models to maintain consistency between all models. Whereas it is consent that support for abstract-to-concrete and concrete-to-abstract (reverse engineering) as well as various entry points should be supported by model-based approaches, tools and approaches that realize these options are still missing. [Florins, 2006] describes the propagation of modifications as tricky, but proposes to determine the side effects on the other models entailed by the application of rules.

Proprietary models Most model based approaches have been strongly tied to their associated model-based system and cannot be exported or are not publicly available. UsiXML [Limbourg et al., 2004b] was the first completely available model format description. Whereas it is currently target to standardization efforts, there is still a generally accepted set of model abstractions missing. Further on, model syntax has been explicitly specified in the UsiXML specification documents, but clear model semantics have not been sufficiently specified so far.

Efficiency and Performance Efficiency and performance of model-based systems are rarely considered. Efficiency needs to be measured for the development cycle implementing multi-platform user interfaces. Performance has to be evaluated at run-time to test if the various supported adaptations do not restrict the user's performance.

2.3 Terms and definitions

Several terms and definitions are relevant for this thesis. In this section we present the most important terms and definitions that are either frequently used in the following chapters or are used to classify and delimit the contribution of this work.

2.3.1 Direct manipulation

Direct manipulation depends on a visual representation of the objects and actions of interest, physical actions or pointing instead of complex syntax, and rapid incremental reversible operations whose effect on the object of interest is immediately visible. This strategy can lead to user interfaces that are comprehensible, predictable and controllable. Direct manipulation programming is an alternative to the agent scenarios. Agent promoters believe that the computer can automatically ascertain the user's intentions or take action based on a vague statement of goals [Shneiderman and Maes, 1997].

This work will focus on generating user interfaces that implement direct manipulation capabilities. Whereas we will follow an agent-based architectural approach, agent based promoters as a specialized form of an user interface will not be addressed by this thesis.

2.3.2 Adaptability and Adaptivity

[Totterdell and Boyle, 1990] distinguishes two complementary properties of modeling adaptation: adaptability and adaptivity. Whereas the former is the capacity of a system to allow users to customize their system from a predefined set of parameters, the latter describes the capacity of the system to automatically perform adaptations without a deliberate user request. Independent of whether the system or the user triggers the adaptation process, [Coutaz and Thevenin, 1999] have identified three orthogonal additional axes that are included into the design space of adaptation: target, means and time. By targeting the environment, the user and the physical characteristics of interaction devices are distinguished. By means of adaptation they denote the software components of a system that are involved in an adaptation. These components are: (a) the system task model that corresponds to the user task model and has been specified by expert on human factors. (b) Rendering techniques denoting the observable presentation and behaviour of the system. And finally (c) the help subsystems that include help about the system and the user's tasks. By the temporal axes they distinguish between static adaptations that are effective between sessions and dynamic adaptations that occur at run-time.

In this thesis we concentrate our work on adaptivity mechanisms that are specifically targeted to the physical characteristics of various interaction devices. Adaptability and the user are not explicitly targeted. Thus the relevant software components that we aim at are the system task model and rendering techniques describing the presentation and behaviour of a multi-platform user interface. The adaptivity of an interactive system at run-time is our main focus but as we although aim at engineering methods for modeling of predefined adaptation scenarios.

2.3.3 Plasticity and Multi-targeting

The term Plasticity is inspired by the property of materials that are able to expand and contract under natural constraints without breaking and provide continuous usage. The term has been first introduced and applied to HCI by [Calvary et al., 2001, Calvary et al., 2004]. In HCI it describes the "capacity of an interactive application to withstand variations of context-of-use while preserving usability". They note that Plasticity is not only about condensing and expanding information according to the context-of-use. But it also covers the contraction and expansion of the set of tasks in order to preserve usability.

Different to Plasticity, Multi-targeting focuses on the technical aspects of adaptation and does not express a requirement in terms of usability, but both terms address the diversity of context-of-use adaptations.

2.3.4 Multimodal Systems and the Fusion/Fission Problem

In general a multimodal system is characterized by supporting several ways of communication with a user by using various modalities such as voice, gesture, or typing. Literary “multi” refers to “more than one” and the term “modal” covers both, the notation of “mode” as well as “modality” [Nigay and Coutaz, 1993]:

“Modality refers to the type of communication channel used to convey or acquire information. It also covers the way an idea is expressed or perceived, or the manner an action is performed. Mode refers to a state that determines the way information is interpreted to extract or convey meaning.”

A multimodal system can take advantage of combining modalities in a sequential or a parallel way. Whereas the former one forces the user to use one modality after another, the latter one allows the user to employ multiple modalities simultaneously. The parallel utilization of modalities results in managing the Fusion/Fission problem to efficiently handle and combine simultaneously incoming information chunks. Whereas Fusion describes the combination of several chunks of information in order to form new chunks, Fission refers to the decomposition phenomenon. Both are part of the abstraction and materialization phenomena [Coutaz et al., 1995].

The fusion/fission problem is out of scope of this work, and so the simultaneous handling of various modalities. As we are focusing on software engineering methods that enable the specification of models to derive user interface for several devices, multimodal systems are not explicitly addressed in this thesis. However, we support developing user interfaces for the heterogeneous landscape of end devices and are targeting our approach to include deriving user interfaces to support several modalities like deriving voice or pen-controlled interfaces.

2.3.5 Context-of-Use

[Calvary et al., 2001, Calvary et al., 2003] define the context-of-use for an interactive system by three classes of physical entities:

1. The users of the interactive system who are intended to use (or effectively use) the system.
2. The physical environment where the interaction takes place.
3. The physical and software platform(s), that is, the computational device(s) used for interacting with the system.

Under software engineering considerations two kinds can be distinguished [Calvary et al., 2003]: Predictive contexts-of-use that are foreseen at design-time. Effective contexts-of-use that are really occur at run-time.

Since we are focusing on specifying a method for developing multi-platform user interfaces, we will specifically concern about the physical and software platforms that should be considered by the interactive system. Both, the predictive context-of-use, we handle at design-time, as well as the effective context-of-use at run-time are addressed in this

work. Addressing the user's preferences or specifically addressing the user's environment is out of scope of this thesis.

2.3.6 Graceful Degradation

The method of Graceful Degradation addresses the trade-off between continuity and adaptation and has been introduced first for safety-critical systems like for instance systems in an aerospace. There Graceful Degradation describes the ability of a system to continue its services proportionally to its components that fail or are malfunctioning. Thus, under bad conditions the system is not expected to fail completely but continue providing at least its basic functions that are offered by all the non-defect components.

Graceful Degradation can be applied to user interfaces as well [Chu et al., 2004]. There it consists of specifying the user interface for the least constrained platform first and then it requires to apply transformation rules to the original interface to produce interfaces to the more constrained platforms. These transformation rules include: Splitting rules, interactor and image transformation rules, moving and resizing rules to reshuffle the user interface, and removal rules [Florins, 2006].

We will follow the idea of the Graceful Degradation by proposing a method that includes capturing all of the interactive systems requirements first. Thereafter we design the overall interactive system's process without initially looking at the platforms constraints, as this might limit the level of freedom of the designer when designing the systems tasks and concepts. Different to following a sequential approach by applying these rules depending on the level of abstraction of the source model during design-time like proposed by IdealXML [Florins et al., 2006a], we follow a more flexible approach and apply the transformation rules at run-time of the system and for all relevant levels of abstraction at once. This approach will be discussed in detail in section 4.2.5.

Beneath the major terms and definitions that have been introduced and classified in the context of this thesis, some terms for which no ambiguous definition exist, will be used as synonyms. In the following two of those terms that we will extensively use in the next chapters will be shortly presented.

2.3.7 Platform vs. (End-) Device

Up to now, a user that needs to interact with an interactive system requires at least one end-device that technically connects her to the system. Whereas an end-device consists of a (physical) hardware platform and at least one software platform, the latter one describes a system that is able to interpret a specific user interface description language and to present it to the user. For this thesis we always refer to the software platform part of a device when talking about a platform or a device.

2.3.8 Designer vs. Developer

Whereas a designer is typically the one that illustrates the layout of a user interface, the role of a developer is to design and implement the interactive system conforming to the design guidelines that have been identified and specified by the designer. For this work

2 Fundamentals

we use the developer role and the designer role as synonymous, because both should be able to follow the proposed method for implementing multi-platform user interfaces.

3 State Of The Art

This chapter analyses related work in the area of interactive system development and is organized into three parts relevant for the contributions of this work. First we take a look at methodologies and tools supporting the design and development of interactive systems in section 3.1. Thereafter, in section 3.2 we present architectural paradigms and patterns that are currently utilized and finally, in section 3.3 we give an overview about the most relevant user interface description languages to specify and design interactive applications.

3.1 Methodologies and Tools for Developing User Interfaces

User interface development has been defined to conform traditional development models for software engineering. The Waterfall model for instance connects several phases into a pipeline where all prerequisite work for each phase is undertaken before that phase starts. This model primarily emphasizes cost estimation and control, as it requires the work to be undertaken in a realistic order. The spiral model reduces some limitations of the Waterfall model by enabling iterative cycling through the phases [Boehm, 1988]. An alternative development model is the V-model, which not only relates each phase to its predecessor and successor, but also demands for a construction and testing phase on the same level of detail [Gram and Cockton, 1997]. The V-Model has been extended to support backtracking, which enables to jump back one or more phases. Each backtracking step results in recovery work to extend or correct inadequacies of the previous phase. The basic idea of evolutionary prototyping is to undertake all phases of the V-model for just one function at a time [Davis and Bersoff, 1991]. The development of features and functions in evolutionary prototyping is ordered by priority and starts with those functions with the highest importance or anticipated stability. Different to evolutionary prototyping, rapid prototyping results in throw-away prototypes through quick runs through all phases of the V-Model to evaluate ideas and to test hypotheses. Participative development has been proposed as one way trying to 'foresee the unforeseen' [Gram and Cockton, 1997] and involves users in all design phases [Muller and Kuhn, 1993].

System development can also be analysed based on the human roles within it. The division of work into several objectives that are associated with a certain role is compatible with the assumption to separate the development of interactive systems into a user interface and the functional core [Gram and Cockton, 1997]. Roles are defined specifically to design a particular level of abstraction. [Gram and Cockton, 1997] lists several roles that are relevant for interactive system development, including a client, a project manager, a user representative, a requirements specialist, a system designer, an implementor, a validator, a user, and a system administrator.

3 State Of The Art

Nowadays development processes are required to be much more flexible as specified by the traditional methods like the V-Model or the Spiral Model to be used in organizations. This is because of rapid changes of today's organizations and their businesses, which need to be addressed by a development process that can manage changing requirements. Changes include, but are not limited to task reallocation among workers, support for redefinition of the organizational structure, and multi-lingual user interfaces required by world-spanning organizations. Workers demand for more usable user interfaces that support new platforms, the migration of interactive applications from stationary to mobile devices, task transfer from one user to another one, adaptation to dynamic environments and redesign due to obsolescence [Limbourg et al., 2004b].

Software development methods have to consider that organizations react to these changes very differently. Regarding interactive applications some organizations follow an *bottom-up approach* starting by recovering and redrawing existing screens and continuously advancing the functional core after the customer has validated the new user interface. Other organizations start by refinements of the task and domain models to be mapped further to screen design (*top-down approach*), whereas some apply all modifications simultaneously where they occur (*wide spreading approach*) or rely on an intermediate model and proceed modifying task, domain and the final user interface in parallel (*middle-out approach*) [Limbourg et al., 2004b].

Following [Vanderdonckt and Berquin, 1999], four major approaches for a process of user interface development can be observed:

1. The *traditional approach* relies on a graphical user interface editor that is used to select interaction objects from a palette of widgets. The developer usually implements the user interface by dragging and dropping interaction objects from the palette to the workspace surface and progressively enhances the user interface representation in this way. In parallel to this process of evolving the graphical user interface presentation the developer can switch to a source code view in order to add behavior by adding semantic function calls to complete the interactive system.
2. In the *programming by demonstration approach* the developer uses an example of various parts of the final user interface to demonstrate the look and behavior of the resulting interactive system. For each exemplary user interface part a set of actions can be associated enabling the reproduction of the example in future user interfaces. Examples that are implemented once can be reused over time as they can serve for many different user interfaces.
3. *Model-based approaches* start by modeling the concepts in a graphical model editor. Typical models that are designed at the beginning of a model-based approach are an application model, a domain model or a data flow model. These models build the foundation for the automated generation of the user interface without the need for human intervention. In case that the developer is more involved in the user interface generation process by for instance explicitly accepting or neglecting design decisions the process is called *computer-aided design*.

4. *Task-based approaches* implement a process very similar to model-based processes except that they start by a task-analysis step. The other models can then be specified, derived or refined depending on the task model.

In the following sections the most important approaches for development processes of interactive systems will be discussed. Starting with the traditional user interface development processes in section 3.1.1, we take on the one hand a short look back to the beginnings of the user interface development and on the other hand describe how these traditional approaches still remain in current user interface design. Section 3.1.2 presents the programming by example approaches that we will grasp upon later again in the design of our methodology in chapter 4. Finally we describe the foundations of model-based and task-approaches in section 3.1.3.

3.1.1 Traditional interactive system development

Traditional user interface development decouples the user interface implementation from the functional core development. Whereas the latter one often follows a well-structured traditional software engineering method, the former one is most often realized separately by iteratively designing non-functional mock-ups and only anchored in the implementation phase of the overall development process.

This separation has several advantages, as multiple user interfaces can be developed for one application, designs can be rapidly prototyped and implemented before the application code is written, and changes that have been discovered through user testing can be incorporated more easily.

Over the years the advantages of strict separation figured out to be not working out as expected. First, the decoupling between front-end and back-end required a huge amount of callback-operations and introduced an additional layer of complexity between both components. Second, with the growing amount of different platforms that are required to be addressed, the user interface consistency has to be ensured on the one hand between all these platforms and on the other hand has to be maintained not only for single screens but for whole processes consisting of several screens.

[Myers, 1995] has identified four different kinds of tools that are applied for direct graphical specification of user interfaces: Prototyping tools, card sequencers, interface builders, and editors for application specific graphics. All these tools have in common that they require the developer to place the user interface objects on the screen by using a pointing device.

Prototyping tools enable a designer to quickly mock up examples and to show how some aspects will look. Most of these prototyping tools like for instance Director, Photoshop, or Visio can only paint or sketch interfaces but can not produce operable interfaces as no real widgets are used. The main advantage of prototyping tools lies in the fact that they do not require programming skills and can be efficiently used by designers.

Card based systems can be used to produce sequences of mostly static pages where each page consists of a set of widgets. Usually a fixed set of widgets is supported by a card-based system to conform the windowing systems widget set. Card-based system

like for instance HyperCard from Apple offer the advantage of allowing to realize a more dynamic interface prototype that often can be executed by interpretation but are limited to support only a fixed set of widgets.

Interface builders also rely on a fixed widget set, which enables dialogue sequencing, and the creation of sub-dialogues. The construction of a user interface is mostly done by following a drag-and-drop interaction style allowing the designer to easily layout user interface screens and to specify fixed positions for all user interface elements. Since interface builders can produce code, it is usually important that the programmer does not edit the produced code as this will prevent the tool to be used for later modifications. Popular interface builders are offered by VisualBasic from Microsoft and by the Netbeans GUIBuilder from SUN Microsystems.

Finally, data visualization tools emphasize the display of dynamically changing data and are used as front ends for simulations, process control, network management, and data analysis. Similar to interface builders, data visualization tools like for instance DataViews from V.I.Corp. rely on a set of widgets, which is specific to the visualization domain, but does not conform to a toolkit.

By using a tool, the produced applications are expected to offer more consistent user interfaces. Further on, experts like for instance psychologists or human factor specialists can be easier involved in the design process. This is because the level of programming expertise can be lower as a tool can hide much of the complexities of creating user interfaces.

3.1.2 Programming by Demonstration / User Interface Prototyping

Prototyping gained a lot of attention in the development of highly interactive graphical applications. This is because the acceptance of such system depends on a large degree on the quality of the user interface at an early stage. [Bäumer et al., 1996] notes that in recent years a lot of projects changed from traditional life cycle or waterfall approaches to prototyping as part of a deliberate evolutionary strategy on the operative level. They distinguished four different types of user interface prototypes: *Presentation Prototypes* illustrate how an interactive system can solve given requirements and for this they are strongly focusing on the user interface, thus they implement a *horizontal prototyping* approach. *Functional Prototypes* are implementing strategical important parts of the user interface. Functional prototypes are realizing a *vertical prototyping* approach and are not only focusing on the user interface but implement the required application functionality as well. *Breadboards* are used to concentrate on technical aspects like the system architecture or specific functionality of the planned interactive application and are not intended to be evaluated by the user. Instead they investigate certain aspects of a special risk. *Pilot systems* are matured prototypes that can be practically applied.

A widely accepted approach for classifying prototyping has been proposed by Christiane Floyd [Floyd, 1984]. She suggested the distinction between exploratory, experimental and evolutionary prototypes. *Exploratory Prototyping* is used to clarify requirements and potential solutions. Related to task analysis it can be used to illustrate the achievements of a task supported by a computer. *Experimental Prototyping* concentrates more

on the technical aspects of selected requirements and is used to prove the feasibility and suitability of a particular implementation. Finally *Evolutionary Prototyping* describes a continuous process for enhancing an interactive system to potential changing requirements.

3.1.3 Model-based and task-based approaches

Models can help to drive the design and development of interactive systems as with the help of models one can manage complexity. Each model introduces a level of abstraction of a real world entity. Abstraction is a very natural way that people usually like to do to manage too complex entities: They start to discover the main aspects that should be taken into account as well as their relations.

The development of an interactive system is a complex problem that is currently continuously growing with the evolving technology enabling to consider the context-of-use during interaction or by enabling multi-modal access to an application. Model-based approaches can be helpful to manage this complexity. Models can be described on different levels of formality but offer a structured description of relevant information.

In the model-based community there is a common understanding that a complex system, like an interactive system can be based on a small set of clear basic concepts. Thus, model-based approaches consider various possible viewpoints over an interactive system. Such viewpoints can be made on different levels of abstraction as well as they can have a different focus (if the user interface or a task is considered). Typical abstractions are [Paternò, 2005]:

- The task and object model on a conceptual level. Tasks are used to describe the logical activities to reach a user's goal. An typical approach is to structure this tasks (for instance by using a task hierarchy) and to relate them by temporal relations. Tasks are associated to objects that need to get manipulated to perform a task.
- An Abstract User Interface is used to describe the logical structure without targeting it to a specific modality or even a platform. An Abstract User Interface refers to interaction objects that are described in terms of abstract interactors.
- The Concrete User Interface that replaces each abstract interactor with a concrete interaction object that is dependent of a modality and a set of associated platforms that implement the same modality and are similar regarding their capabilities (for instance the screen size, and the control features).
- A Final User Interface the has been transformed from the concrete interface and runs on a specific environment and language (for instance Java or HTML).

By a model-based design the developer has to introduce the complete specifications of the underlying models to produce a first user interface that he refines for a specific context later on. More detailed introductions and discussions about model-based development approaches can be found in [Balbo et al., 2004], [Paternò, 2005] and [Szekely, 1996] respectively.

Other approaches: Object-oriented modeling and cognitive models Beneath the model-based approaches that will be discussed in the next sections, other types of models exist that can be used for the design of an interactive application.

On the one hand there exist a broad range of object-oriented modeling methods. One of the most successful is the Unified Modeling Language (UML), which shares partly the same purposes like task-driven approaches as it contains both, the description of static structures as well as modeling dynamic aspects like for instance activities. Differences between both approaches can be found in the focus and the notations used to describe a system. Different to task-based approaches, object-oriented techniques focus on the identification of objects and their compositions first, whereas task-based approaches start by identifying the activities and thereafter continue describing the manipulation of objects associated to these activities.

[Paternò, 2005] describes task-based approaches to be more suitable to design user-oriented interactive systems, because of their main focus to effectively and efficiently supporting the users' goals. This conforms to an important usability principle, which postulates to focus on the user and their tasks [Paternò, 2005]. Different to task-based approaches, object-oriented software engineering techniques can be applied more powerful at the software implementation level. Disregarding the different focus of both approaches and looking at the specific techniques used in UML there exist some similarities. Especially activity diagrams have very similar purposes like task notations but illustrate multiple levels of abstractions only in a very limited way. Use cases address the requirements analysis phase like the output of task analysis. There have been also some work transforming sequence diagrams into task hierarchies by removing the internal system oriented view [Lu et al., 1999] but the main reason to support these approaches lies in the general acceptance and knowledge of the UML notation and the pre-existence of tools that can be considered for task analysis.

On the other hand there have been various cognitive models proposed focusing on an integrated description of human-cognitive mechanisms, interactions between these mechanisms, and their simulation by automatic tools. Modeling techniques can be distinguished into two different types: descriptive and prescriptive models. A descriptive model describes the history of how a particular software system was developed. Descriptive models may be used as the basis for understanding and improving software development processes, or for building empirically grounded prescriptive models [Scacchi, 2002]. A lot of cognitive models are descriptive models used for task analysis targeted to characterize and identify tasks. Good examples are TKS, MAD [Rodríguez and Scapin, 1997] or GTA [Veer et al., 1996] and GOMS [Card et al., 1983] or for performance evaluation CPM-GOMS [Gray et al., 1992] or KLM [Card et al., 1980]. Those approaches are beyond the scope of these thesis, which concentrates on finding a notation suitable to represent tasks and their relationships more precisely. Their representation offer recording paths of user actions including both, motor steps and mental steps in performing tasks. These approaches can be used to evaluate interface design and user performance.

In the following the relevant steps of the last decade of model-based user interface development will be shortly reviewed focusing on the major results that are broadly used until these times and are relevant for the derivation of our requirements in chapter 4.

Other relevant contributions during the same period have been reduced to note their special benefits and shortcomings.

3.1.3.1 First generation: Declarative Descriptions

The first generation started with first works at Jim Foley's research group in 1994. Together with his colleagues he proposed the *User Interface Development Environment* (UIDE) [Foley et al., 1988, Foley and Kovacevic, 1988, de Baar et al., 1992, Byrne et al., 1994] at the Gvu Center of Georgia Tech. UIDE enables the developer to specify pre- and post-conditions for each interactor of an interactive system. Post-conditions are used to modify the interactor after an operation has been performed by the user through the user interface and therefore reflect state changes. Pre-conditions enable an interactor and make it active for user interaction. UIDE was able to reason about the interactor model and can critique a given design for consistency and completeness like proposed in [Braudes, 1990]. Further on a sequence of animated help can be automatically derived using the pre- and post-conditions during run-time of the models in case UIDE detects end-users difficulties during their interaction with the UIDE system [Sukaviriya and Foley, 1990].

Humanoid proposed by Szekeley et al. in 1992 [Szekely et al., 1992, Szekely et al., 1993] designed a declarative modeling language that consists of five independent parts: the *application semantics*, the *presentation*, the *behavior*, the *dialog sequencing* and the *action side effects*. The purpose of the Humanoid approach was to separate the design information entered by a user interface design tool from the tool itself in a way that changing requirements for the user interface can be explored by adding new user interface designs and by adding new code. The application semantics define the domain of discourse for the interactive system and is referenced by the other dimensions. The presentation defines the visual appearance of the user interfaces, whereas the behavior dimension combines the definition of the possible input commands (gestures) that can be applied to manipulate the state of the application and the corresponding presentations. The Dialog sequencing dimension defines the ordering of how commands can be executed through constraints. Finally the action side-effects dimension specifies actions that can occur automatically if the application state changes or the user inputs a command.

Humanoid can be characterized as an top-down approach as it lets designers express an abstract conceptualization of an interface design based on five dimensions that can be refined step by step. The introduction of the specification of five independent dimensions allowed designers to explore the design space in any order and without requiring a fully concretized application model. Humanoid's methodology (depicted in figure 3.1) is an iterative and sequential process. The process starts with the Application Semantics Design that can be done in parallel with the Presentation Model Design. The Manipulation Specification is linked to the Application Semantics and the Presentation Model in the next step. Finally, in the last step, the Sequencing can be defined that is entirely dependent on the Manipulation Specification whereas the action side-effects link the Manipulation Specification with their side-effects to the Presentation Model.

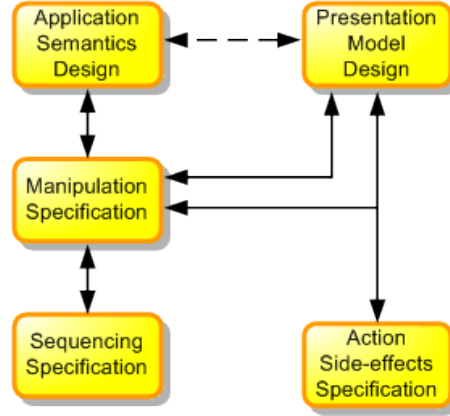


Figure 3.1: Humanoid Interface Development Methodology.

Humanoid improved the UIDE by providing a richer model of command states and therefore enabling designers a much finer control over the dialogue sequencing. Both, Humanoid and UIDE are targeted to support the designer by offering explorative designs. Other early model-based approaches like APT [Mackinlay, 1986], Mike [Olsen, 1986], SAGE [Roth and Mattis, 1990], and UofA* [Singh and Green, 1989] focus on the automation of the interface generation. Janus [Balzert et al., 1996] and Mercano [Puerta, 1996] concentrated on modeling the underlying application domain using a domain model. Such very limited views of the modeling domain have been typically used to produce simple menu or form-based interfaces.

The Managing Interface Design via Agendas and Scenarios (MIDAS) approach by Lou [Luo, 1994] was one of the first attempts that aims to find an adequate balance between high-level automation approaches and offering designers extensive control over design decisions. Therefore MIDAS offers four basic components: Humanoid's explicit interfaces model, an interactive and iterative modeling system, a library of design goals and a management system that maintains an agenda about the status of the design and development process.

3.1.3.2 Second Generation: Consideration of Task Models

A task is an activity that should be performed to reach a goal. A *goal* is a desired modification of state or an inquiry to obtain information about the current state of an object or system [Mori et al., 2002]. A task model allows a hierarchical structure of tasks that can be performed by the user, the system or both. The introduction of a task based design and analysis was accepted to focus on a more user-centric design philosophy compared to the earlier system-oriented perspective. Thus, a task model can be seen as the knowledge source where the user interface is described in the user's own words, not in terms of the system or designer [Wilson and Johnson, 1995]. [Paternò et al., 1998] notes, that task models are accepted to be highly expressive and since task modeling typically starts during the analysis and design phase, all the derived models can be expected to

be expressive as well.

Procedures that realize system functionality are located at the domain model. Thus, domain model and task model are typically strongly related to each other. Beneath the domain and task model other models have been added to support model-based user interface development: The user model specifies characteristics of the intended users of the interactive system. Different groups of users can be categorized and tailored with personalized user interfaces. The user model can include user capabilities, psycho-motoric skills, their language and culture. A dialogue system is used to refine the task model and provide richer interfaces that more closely reflect the navigation and interaction principles the designer desired. Finally a presentation model has been proposed to specify the characteristics of the presentation in a more fine grained way. Widgets and dynamic facets of the interface can be more specifically refined using a presentation model.

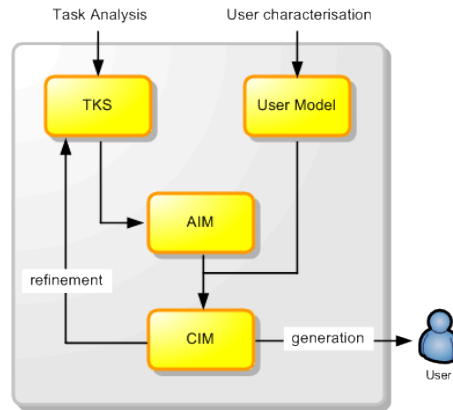


Figure 3.2: The Adept Interface Development Methodology.

The Advanced Design Method for Prototyping with Taskmodels (ADEPT) [Johnson, 1991, Johnson, 1992, Johnson et al., 1993, Johnson et al., 1995] was one of the first approaches to include a task-model in the development process (like depicted in figure 3.2). The general agreement that task models have to be included in the development process founded the second generation of user development processes relying on model-based approaches around 1993. In the ADEPT approach an abstract architectural model (AIM) is designed that is based on a task model (TKS). But ADEPT only offers a limited environment, which does not consider adding new interface styles or give designers enough control over the interface details. [Wilson and Johnson, 1996] mention the generally missing identification and description of design activities that guide the designer from task analysis to task modeling of interactive applications and proposes initial guidelines and design activities to complement the ADEPT approach as well as present several implications for an efficient tools-support.

In the Tools for an Interactive Development Environment (TRIDENT) project [Bodard et al., 1994, Bodart et al., 1995a, Bodart et al., 1995b, Vanderdonckt and Bodart, 1993] the user-task modeling has been combined with a knowledge base of interface design guidelines. This approach enabled the generation

3 State Of The Art

of high-quality user interfaces, because the knowledge base can be analyzed during the generation process. TRIDENT relies on an Activity Chaining Graph that describes the data flow between the application domain functions to specify the task model. The domain model of TRIDENT is structured using entity-relationship modeling. Different to other approaches TRIDENT focused on the generation of form-based user interfaces only.

Mastermind [Szekely et al., 1995] combines the design-time environment with a run-time environment that is used for the delivery of the user interface. The design-time architecture consists of three models that can be developed independent from each others: The application model defining the capabilities of the application and implemented as an extension to the CORBA object model. A task model describes the tasks the user can perform within the interactive system and outlines the required steps to perform these tasks. The task models of Mastermind are similar to those of GOMS, TAKD and UAN but can be used to drive the user interface at run-time as well, rather than just specifying them. Mastermind uses pre- and postconditions to model the dialogue component of the user interface similar to UIDE. The presentation model is specified using a specialized modeling language that concentrates on specification of layouts via a constraint-based system. All three models are based on separate, specialized modeling languages. Mastermind supports a technique to compose these models at run-time [Stirewalt, 1999, Stirewalt and Rugaber, 2000]. Using a language for binding the user-interface models, inter-model relationships can be specified to generate the final user interface. As the models are developed separately from each others the problem of model-to-model transformation was out of the scope of mastermind.

The Object Action Interface (OAI) Model [Shneiderman and Plaisant, 2004] has a strong focus on building direct manipulation interfaces. It can be used at design-time to capture an interface from an object-oriented perspective. Following the OAI approach, the designer starts by identifying the tasks that the system has to perform. In OAI real-world objects and actions are correlated to corresponding interface objects and actions. Therefore tasks are refined into single steps, each realizing an action that has to be performed by an object. In OAI temporal relations cannot be captured and especially voice interfaces that require a sequential processing of interaction steps cannot be realized. The MUSE method [Lim and Long, 1994] offers three main development phases: during the information elicitation and analysis phase by using task analysis techniques the designer collects background design information that relates to both, the system currently in use and other related ones. The first phase ends with a Generalized Task Model, which provides a device-independent task model of the existing system. Whereas the main objective of the first phase is to identify problems, in the second phase, the design synthesis phase, a Composite Task Model (CTM) is developed, which is divided into a system and a user task model. Finally, in the design specification phase, an interaction task model is designed, which provides the device-level actions to be performed by the user. Additional pictorial screen layouts are defined corresponding to the specification of the CTM.

3.1.3.3 Third Generation: Incorporated Tools

By incorporating tools into the task-based design process some of the limitations of the first and second generation of user interface development have been weakened. In the first generation mainly engineering-centric tools have been proposed that concentrated on windows and widgets. In the second generation the user-centric design approach by a methodology has been recognized as critical to design effective user interfaces. In the third generation both approaches have been combined to form an integrated user-centric and tools-supported environment.

The Model-Based Interface Designer (MOBI-D) [Puerta, 1997, Puerta and Eisenstein, 1999] was one of the first proposals for an integrated model-based interface development environment that embraces the idea of designing and developing user interfaces by creating interface models. MOBI-D included model-editing tools, user-task elicitation tools, and an interface building tool called Model-Based Interface Layout Editor (MOBILE) [Puerta et al., 1999]. MOBILE includes a knowledge-based decision support system to assist the designer. By a knowledge base of interface guidelines an inference mechanism traverses the attributes and objects of the domain model to propose suitable widgets to the designer.

The Task-based Development of User Interface Software (TADEUS) approach [Schlungbaum, 1997] proposes dialogue models as the central aspect when designing interactive systems and clearly separates the task model specification from the dialogue model. Therefore TADEUS divides the user interface development approach into three stages. First a requirements analysis is done based on three domain models (task, problem domain, and user model), second by the dialog design the static layout and the dynamic behavior of the application is specified, and finally a prototype is automatically generated.

The Teallach tool [Griffiths et al., 1998b, Griffiths et al., 1999, Gray et al., 1998] extends these approaches to include comprehensive facilities for relating the different design models and offers a design method in which models can be constructed and related by designers in different orders and in different ways. As TADEUS for instance provides a systematic approach to the refinement of models, it encourages a particular sequence in the specification of interface designs, which is not required in Teallach where it is not necessary to start with a task model. Thus, tasks that get derived later on are not user-centric, as [Vanderdonckt et al., 2000] notes. This problem can be avoided through the definition of domain-specific derivation rules (for example in the domain of data-intensive systems like described by [Barclay et al., 1999]).

The ConcurTaskTree (CTT) notation is one of the most cited approaches for tool-based task analysis and design of interactive applications. It has been designed to offer a graphical syntax that is easy to interpret and designed by using a tool, the CTT Editor. CTT can reflect the logical structure of an interactive system in a tree-like form based on a formal notation. Different to other notations like the User Action Notation [Hartson and Gray, 1992] the CTT notation abstracts from system-related aspects to avoid a representation of implementation details. [Paternò, 1999] states that a compact and understandable representation was one of the most important design aspects of CTT

3 State Of The Art

in order to enable the modeling of rather large task models for industrial applications in a compact and easy reviewable way even by people without a formal background.





	<p>A <i>user task</i> illustrates important cognitive processes performed by the user. Example task types of these category are <i>planning activities</i> to organize a sequence of actions to perform, <i>comparing activities</i> by evaluating informations like for instance comparing quantities or <i>problem solving activities</i>.</p>
	<p>An <i>application task</i> is completely executed by the computer without any human intervention through internal processing. The typical purpose of an application task is to present results of a computation to the user. Application tasks generate overviews by <i>summarizing</i> data, <i>comparisons</i> assist the user by preparing a presentation containing required data, <i>locating</i> tasks enable the user to rapidly find things, whereas <i>grouping</i> clusters information to be presented at the same time. Finally <i>feedback processing</i> prepares information to the user without requiring a specific request from the user.</p>
	<p><i>Interaction tasks</i> describe tasks realizing some interaction techniques to enable the user interacting with the system. Interaction tasks include <i>selecting</i> objects, <i>editing</i> data or enabling the user to <i>control</i> some actions explicitly through events.</p>
	<p><i>Abstract tasks</i> specify complex activities that cannot be unambiguously specified. Thus tasks who's children tasks belong to different categories have to be set to abstract as well.</p>

Table 3.1: Task categories of the ConcurTaskTree notation.

A CTT tree represents a hierarchy of tasks, each categorized in one of four categories (see table 3.1). Each layer of the tree refines the level of abstraction of the tasks until the task tree leaf nodes that are called basic tasks and cannot be refined any further. Tasks that have the same parent task can be combined using temporal operators to indicate their relationships.

An often mentioned downside of CTT is, that it requires to introduce artificial parental tasks to avoid ambiguities in the temporal constraints. These super tasks do not have any value for the user and make the model harder to understand. Further on, the support

to model conditions is very limited and not explicitly available in the CTT notation and non-temporal relations between tasks are not supported (for instance the interaction that has to be entered to perform a task might depend on the data entered in a previous task). Finally the support of CTT for incremental modeling is limited as often semantics cannot be added by just editing one place but often require the addition of new tasks.

3.1.3.4 Fourth generation: Tool-supported multi-path, multi-context development

After tools have been incorporated into a user-centric design process in the third generation, addressing the various contexts-of-use and support for multi-path development is the main target of research in the fourth generation. Whereas the former is targeted to support user interface adaptations for combinations of a certain platform, user, and environment, the latter is about identifying and connecting various levels of abstraction to design an application.

The Cameleon Reference Framework defines several design steps for the development of multi-context interactive applications. The basic development process is divided into four steps and is illustrated for two contexts of use in figure 3.3. The Cameleon Reference Framework describes every development step as a manipulation of the artefacts of interests in form of a model or a user interface representation [Calvary et al., 2003].

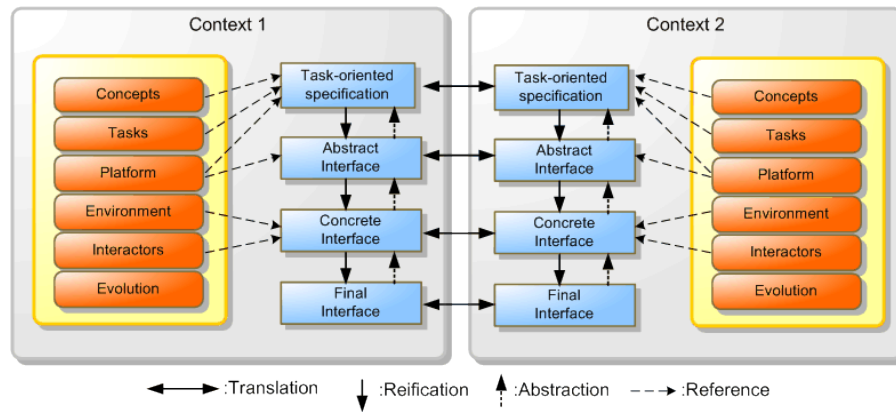


Figure 3.3: Cameleon Reference Framework.

The process starts by a description of *tasks* that have to be carried out by the final interactive system and by a description of the *domain-oriented concepts* that are needed by these tasks. Concepts are represented as classes that get instantiated to objects used to reflect the concept manipulation.

From the task and concepts level an *Abstract User Interface* (AUI) is derived. The AUI groups the subtasks of the task model into interaction spaces (presentation units) according to a set of criteria like task model structural patterns, cognitive work load analysis or identification of the semantic relationships between tasks. Further on the AUI defines a navigation scheme between the interaction spaces and selects Abstract Interaction Objects (AIO) for each of the concepts that have to be independent of any

modality of interaction.

The Concrete User Interface model (CUI) concretizes the AUI and defines the widget layout and interface navigation through these widgets. Different to the AUI, the CUI is designed for a concrete modality but independent of a specific platform. The CUI is used to present the final look and feel of the user interface but has to be considered as a mock-up that only runs within a particular environment. On the lowest level of abstraction the Final User Interface (FUI) represents the operational user interface that can be run on a platform either by interpretation through a renderer (for instance a web-browser) or by execution (after compilation into binary code).

Between these four levels of abstractions three types of transformations can be identified: The Reification transformation is used to concretise abstract artifacts to more concrete ones, whereas Abstraction is a process of generalization of concrete artifacts. Translation is used to transform a certain level of abstraction that is modeled for a specific context-of-use to a different context-of-use without switching the level of abstraction. By applying these transformations a designer can start her development activity from any model of the reference framework.

The multi-path user interface development process based on the Cameleon Reference Framework follows a transformatal approach to map between the four levels of abstraction. The presented models enable to build modular tools that can guide the designer through the whole or only parts of the development process. Up to now Vanderdonkt et al. have developed a whole bunch of tools [Michotte, 2008, Collignon et al., 2008, Coyette and Limbourg, 2006, Montero and López-Jaquero, 2006] that are based on a language (UsiXML) making the models and their associated transformations explicit. The downside of an transformatal approach lies in the complexity of the transformations required to map between the models. This problem is known under the name “mapping problem” and will be introduced in the next section, as it is relevant for all transformational approaches.

Regarding the Cameleon Reference Framework the introduction of arbitrary entry points and the abstraction transformations are target of ongoing research. For instance the abstraction from the abstract to the task model figured out to be complicated and understudied so far. Different to typical software engineering approaches, well separated phases and interactions were not made explicit for the reference framework so far and some phases like for instance “testing” are still missing.

Tools Several tools have been proposed to support the design and specification of the various models required to realize multi-context and multi-path development. Since there is no commonly accepted set of models and a widely accepted method to support model-based development is still missing, the tools differ in the models, notations, development phases and grade of automation they offer.

The Transformation-based Environment for Designing and Developing Multi-Device Interfaces (TERESA) [Berti and Paternò, 2005, Mori et al., 2002, Mori et al., 2003] is composed of a method consisting of four steps that is supported by a tool. First, by high-level task modeling of a multi-context application a single task model is designed

3 State Of The Art

that addresses all possible contexts-of-use, the involved roles, as well as identifies all objects of the domain relevant for performing the tasks. Second, a system task model is developed for all the different platforms that should be supported. The system task model for a certain platform is allowed to refine the task model of the first step. Third, from the system task model an abstract user interface is derived that considers the temporal relationships and composes a set of presentations. Each presentation is structured by the means of interactors. Finally, by the user interface generation a final user interface presentation is generated for all platforms. The TERESA tool supports all these steps and interactively guides the developer through this top-down approach and offers a set of transformations to generate voice and web-based applications.

The Dygimes system [Luyten, 2004, Coninx et al., 2003] (Dynamically Generating Interfaces for Mobile and Embedded Systems) follows the same approach as TERESA and uses a task tree as the initial model to calculate the sets of enabled tasks (ETS). Based on the ETS a dialogue model is derived that forms a state transition network [Luyten et al., 2003a]. In Dygimes a tool supports the designer to attach XML-based user interface descriptions to all atomic tasks of the tree. After the mappings between the task tree and the user interface fragments have been specified, Dygimes offers a tool to describe spatial layout constraints helping the designer to ensure that the interface is rendered in a visually consistent manner. Finally, a run-time library can read the constructed UI specification, which includes the task specification and adapts both to the target platform and renders the user interface based on the calculated ETS.

The Dynamic Model-based User Interface Development (DynaMo-AID) project [Clerckx et al., 2004a] is based on an extended version of CTT task models and supports dynamic context changes. They propose decision nodes and collect distinct sub-trees from which one will be selected at run-time. Further on DynaMo-AID includes a concrete presentation model as well as a description of domain objects. Up to now it supports to generate an application based on UIML [Abrams and Helms, 2002] and SEESCOA XML [den Bergh and Coninx, 2004b]. DynaMo-AID follows a prototype-driven methodology [Clerckx et al., 2006b] that is inspired by the spiral model. It starts from a task model, followed by the generation of a dialogue model. Combined with the concrete presentation model a prototype can be generated, evaluated and refined. [den Bergh, 2006] criticizes that no details are available on the exact notation used by the DynaMo-AID tool for the textual description of the model.

The user interface generation process in DiaTask [Reichard et al., 2004] works similar to DynaMo-AID. It starts with the specification of a task model, which is transformed to a dialogue graph, which is a technique used in the TADEUS approach. It's up to the designer to relate tasks with user interface elements that are represented using XUL¹ as the concrete user interface language. The DiaTask tool supports the animation of user interface prototypes based on the dialogue graphs.

¹The XUL project, <http://www.mozilla.org/projects/xul>, checked 11/29/08

3.1.4 The Mapping Problem

The mapping problem can be seen as a direct consequence following a model-based user interface development approach. The mapping problem has been firstly introduced by Angel Puerta and Jakob Eisenstein [Puerta and Eisenstein, 1999] who state: "if for a given user interface design it is potentially meaningful to map any abstract model element to any concrete one, then we would probably be facing a nearly insurmountable computation problem". Figure 3.4 illustrates the mapping problem between a set of

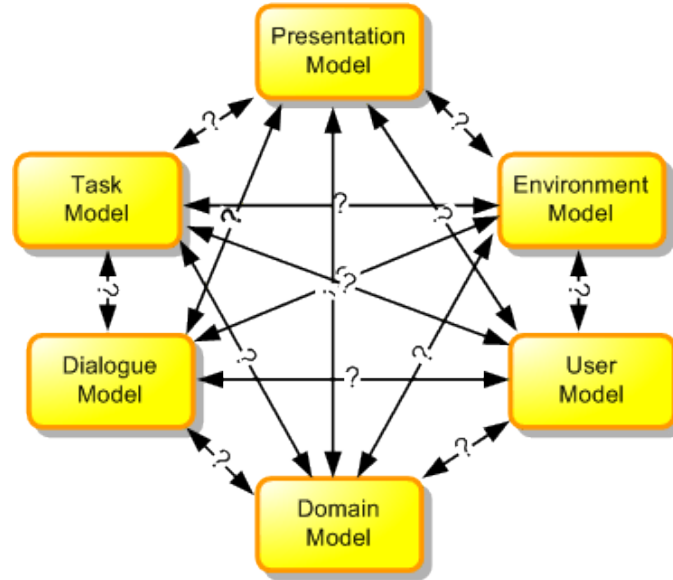


Figure 3.4: The Mapping Problem between models.

models. Solving the mapping problem comprises answering, which models should be combined, at which level of abstraction, and how the models are combined. Mappings specify which interaction objects have to be mapped and how the interaction objects can be linked between different levels of abstraction. Mappings can be maintained in both directions (reification and abstraction).

[Vanderdonckt et al., 2000, Vanderdonckt et al., 2003] describe three mechanisms that can be used to establish mappings between the models: *Model derivation*, *Model linking/binding*, and *Model composition*. Model derivation is a process that uses one or many already specified models to derive one or many unspecified models. This process is typically driven by transformation rules that are interactively applied by a designer that manipulates the transformation parameters [Szekely, 1996, Schlungbaum, 1997, Barclay et al., 1999]. Model linking and binding is a process between already specified models to establish or re-establish relationships between the model elements [Barclay et al., 1999, Brown et al., 1998]. Finally, model composition is used to rebuild the source models (reverse-engineering) or to build new models. This is done by partially or totally assembling already specified models [Stirewalt, 1999].

3 State Of The Art

[Montero et al., 2005] define a formal mapping between the domain, task and abstract user interface elements. Therefore they have realized the IDEALXML tool, which allows to establish a set of mappings either manually or automatically (in combination with the TransformXML tool [Stanciulescu et al., 2005]) based on a mapping model. Since they utilize the UsiXML language [Limbourg and Vanderdonckt, 2004a] for specifying the mapping model, eight different types of mapping relationships can be defined. Beneath the *observes*, *updates*, and *triggers* relationships, that are used to define mappings between an interaction object and a domain model concept, the *isReifiedBy* relationship indicates a reification of an abstract to an concrete interaction object, and the *isAbstractedInto* relationship indicates the opposite mapping direction. Finally, *manipulates* maps a task to a domain concept, *isExecutedIn* maps a task to an interaction object and *hasContext* maps any model element to one or many context-of-uses.

[Clerckx et al., 2004b] extends three mapping mechanisms proposed by [Limbourg et al., 2000, Vanderdonckt et al., 2000, Vanderdonckt et al., 2003] to a classification of five mechanisms to solve the mapping problem. They distinguish between *model derivation*, *partial model derivation*, *model linking*, *model modification*, and *model update*. The DynaMo-AID design process takes advantage of these five mechanisms and directly embeds the model mapping specification into the design process. Whereas their approach mainly focuses on describing the abstract-to-concrete mappings, they identified consistency issues when mapping the opposite way. Especially when deploying a user interface on a constraint device or in cases where the designer wishes to apply changes to the generated user interface, concrete-to-abstract mappings are required to check the consistency with the specified abstract models.

The relationships between the task model and the domain model are analysed in [Pribeanu, 2002] and extended to include the presentation model and tool support in [Pribeanu, 2007]. They identified three different layers of task modeling, namely the functional, the planning, and the operational layer. Whereas mappings on the functional layer are used to map application functions onto user tasks that correspond to business goals, they focus on mappings on the planning layer that specify how a user is planning to accomplish the goal and on mappings defined on the operational layer that show how a user is actually accomplishing a goal with a given technology. Therefore, in [Pribeanu, 2007] three mapping rule types are proposed, two rules have been identified on the lowest layer of abstraction enabling to map horizontally between domain object attributes and abstract interaction objects (rules 1) and between basic control tasks of the task model to abstract interaction objects in the presentation model (rules 2). The third rule is composed of a vertical mapping between tasks of the planning and operational layer and a horizontal mapping between domain objects of the domain model and unit tasks of the planning layer. By using a tool they demonstrate five mappings based on the third rule type that can be used to automatically detail a task tree that has been specified down to the planning layer based on the domain model information and the actual task that should be performed on a certain object (add, edit, delete, display, search).

[Vanderdonckt et al., 2000, Vanderdonckt et al., 2003] investigate into a task-dialogue-presentation mapping and present an approach to derive the navigation of the dialogue

model from the task model's temporal and structural properties. By an interactive tool that utilizes a decision tree, derivation rules can be specified and subsequently applied to the task model to generate the initial dialogue model. The tool uses their own notation, the windows transition notation and supports flexibly changing the generated results but does not consider consistency checks regarding the manual manipulations of the dialogue and presentation models to conform the task model.

3.1.5 Conclusions

The quality of the resulting user interfaces that are developed using a task-based or model-based approach heavily depends on the quality and quantity of information captured in the models itself [Vanderdonckt and Berquin, 1999]. This requires an expressive power of the models and their completeness and consistence. An problematic aspect of model-based approaches is their lack of flexibility and expansibility to consider new interaction techniques, new input and output modalities or even new widget libraries because a user interface produced by a model-based approach is entirely based on the predefined set of input/output (IO) elements supported by the development environment. [Vanderdonckt and Berquin, 1999] generalize the problem of selecting the optimal input and output widgets for a user interface as the "*io selection problem*" and the decision about their optimal placement in the user interface as an "*io positioning problem*". The io selection problem is currently tackled by using production rules (if..then..else constructs) [Foley et al., 1988], selection trees [Bodart et al., 1995a, Szekely et al., 1992], or knowledge trees [de Baar et al., 1992]. All methods end up extending their rules-set or selection trees by refined or extended predefined IO's. As [Vanderdonckt and Berquin, 1999] note, each extension makes these knowledge bases growing, harder to maintain and visualize for the designer.

To consider new interaction techniques in the model-based generation the designer has to directly manipulate the final user interface. If the used development environment is not powerful enough to track and save these manual manipulation the designer is forced to reproduce his changes each time a new version of the user interface is reproduced through the model-based development process.

An often criticised aspect of approaches that start by a task analysis is the additional amount of time that it needs for analyzing peoples' work and representing the most important aspects in a structured model. But construction of models providing temporal and sematic information is a well accepted approach as it is a suitable way to manage complexity especially when designing interactions for very large and complex interactive systems. Another actual problem of task modeling approaches is that they cannot be used for creative tasks as in this special case there is no structured task model used to express the functionality supporting a user in achieving his goals. Finally, some users like to identify the objects first before they focus on the actions, which is not directly supported by actual task analysis.

An extensive comparison of five model-based systems can be found in [Griffiths et al., 1998a]. By conducting an unifying case-study that is about a task for a library application enabling users to search for books, they compare three main criteria:

first, the support for the automatic generation of interfaces; second, the utilization of declarative methods to specify the interface; and finally the adoption of a methodology to support the development of the interface.

3.2 Architecture Models for Interactive Applications

Architecture models are intended to address the needs of interactive applications by separating the concerns assigned to different components within their respective architectures. In this section we give an overview of the most prominent architectural patterns that are currently implemented to support the development of interactive systems. Before discussing the pros and cons of actual architectures, we start by briefly introducing the general interaction model structure. This model is driven by the idea of highlighting the communicative aspect to structure an interactive application by defining interactors in subsection 3.2.1. Thereafter, in subsection 3.2.2 we introduce the separation-of-concerns paradigm, which is the main driving force between all architectural proposals that we discuss in detail in subsection 3.2.3.

3.2.1 Interaction Model

Unlike “classic” computer programs interactive systems have the role of additionally communicating to the user, which leads to a complexity of interactive systems that demands the need for a structure to reduce complexity and enhance maintainability. The communicative aspect is often used as a basis for structuring an interactive system into three domains: input/output definition, application and dialogue sequence, which correspond to the lexical, semantic and syntactic levels [Foley et al., 1990] of linguistic interaction [Shevlin and Neelamkavil, 1991].

Most architectures for interactive systems use the notation of an interactor as the basic interaction component in which applications can be modeled. Therefore, each interactor is dealing with different subsets of human-computer interaction and implemented as a complete information processing system. A set of interactors build a network to structure an interactive system [Coutaz, 1989]. [Paternó and Faconti, 1993] proposed to specify the interactor network structure using a process oriented notation, where each interactor is viewed as an abstract “black box” that mediates between a user-side and an application-side. The process logic is then used to specify the external behavior of an interactor (or interaction object). [Duke and Harrison, 1993, Duke and Harrison, 1995] have proposed a similar interactor model (PIE model) that links states, events, and renderings into an agent-like unit that can serve as a building block for the specification of interactive systems. Different to [Paternó and Faconti, 1993] they use a set-theoretic notation (Z) to define interactors. Each interactor has a presentation state that reflects the internal state of an application.

Beneath these logic-oriented specifications, interactor models can be characterized from an object-oriented perspective since they conform to the observer pattern regarding the communication between the interactors and the composite pattern, which describes the hierarchy of interactors in the network [Gamma et al., 2000].

3.2.2 Separation of Concerns

The separation of concerns principle is a widely accepted concept in the software engineering area to improve re-usability, portability, modifiability and integrability [Bass et al., 1998]. In the field of user interface development this concept has been adapted very early, as a tight coupling of the application logic with the presentation in the early days was a significant barrier in reusing user interfaces [Alencar et al., 1995]. The application logic component usually implements the system logic containing the functional core of the application for instance by accessing external (web-)services, a file system or a database. The presentation component stores the informational data structures that are presented by a user interface to interact with the user. Further on the presentation component makes assumptions about the capabilities of the end-device used by the user to interact with the system (for instance screen resolution, input capabilities, etc.).

[Menkhaus, 2002] notes that research on user interfaces has focused early on establishing reference architectures that are intended to help the developer:

- to build on already established designs,
- to get benefit from learning from the experience of others,
- to not have to reinvent solutions from commonly recurring problems,
- to get a head start on common problems and
- to avoid common mistakes.

Whereas the separated application logic can be executed on a server, via a proxy or on the client device, the user interface must be executed entirely on the client device. This led to the famous *user interface explosion* problem (“M times N problem”), which has been transferred to web pages and devices in [Case et al., 2000]. The user interface explosion problem states that an application exports its functionality via M interfaces that have to be accessed with N devices. This requires the implementation of MxN user interfaces. Looking at other domains this problem has been reduced to the amount of M+N with the introduction of an intermediary step. It is one goal of the architectural model proposed in this thesis to reduce the user interface development effort for multi-platform user interfaces to an M+N problem.

Agent-based Models Over the last decade several models for architectures for interactive systems have been proposed to factor them into different components. A major milestone was the introduction of agent-based models to structure an interactive system as a collection of specialized computational units called agents.

[Coutaz et al., 1995] lists the following benefits using an agent-based style:

- The functional modularity of an agent makes it possible to modify its internal behavior without the need to change the rest of the system.

- An agent can be associated to one thread of the user's activity. Each agent has its own state that it is managing internally, which enables the user to suspend and resume the interaction on the user's choice.
- Complex threads of activity can be easily distributed across a set of co-operating agents.
- As an agent defines its unit for processing, it can be executed on a different processor or machine from where it was created. For the implementation of groupware systems for example, instances of classes of agents can be run on distinct work-spaces.
- Looking at the implementation level, agent models can be easily implemented in terms of object-oriented languages.

3.2.3 Architectures describing Interactive Systems

The following section discusses the major contributions for models that can be used to derive architectures to realize interactive systems. The section starts historically with one of the first contributions in this area, the Seeheim model (section 3.2.3.1) and continues describing and discussing two representative agent-based models, the MVC (section 3.2.3.2) and the PAC-Amodeus (section 3.2.3.4) model. Chronological between these both, the Arch/Slinky model (section 3.2.3.3) has been proposed that was one of the first attempts that considers a meta-model. Finally, the Meta-Interface Model (section 3.2.3.5) will be discussed that focuses on actual problems regarding device-independence. Since there are a lot of more models available that can't be discussed in greater detail, section 3.2.3.6 briefly summarizes the contributions of other important models. The results of these discussions (section 3.2.4) will serve as a foundation for the design of the run-time architecture that will be introduced in chapter 5.2.1.

3.2.3.1 Seeheim

Early approaches focused on two primary goals: firstly to minimize the effects of interface changes on the application as a whole and secondly to promote portability among different windowing systems. The Seeheim model was the first model to focus on a solution to these problems [Green, 1985]. It had been established by X/Open Technology in the eighties when most application logics were accessed through either command line or form filling interfaces.

Figure 3.5 illustrates that the Seeheim model is separated into three layers: The presentation layer is the static and visual part of the interface where all input is entered and all output to the user is performed through interface objects. In contrast to the presentation layer, the dialog layer has a state to structure the user-application dialog. It does so by handling all events and interfaces between the static screens of the presentation layer and the application layer. Every communication between the user and the application layer, which implements the interface to the functional core, runs through the dialog

3 State Of The Art

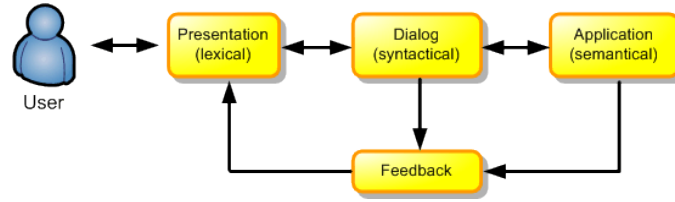


Figure 3.5: The Seeheim Model.

layer. For efficiency reasons a feedback bridge has been introduced; if an application requires to give fast response to user input the feedback bridge can be used to bypass the dialog layer.

[Green, 1985] lists the following advantages arising from the separation into three layers, that can be designed, implemented, tested and maintained independently from each other: Beneath the improved re-usability and portability, implementation tasks can be easier distributed among the interface designers and application developers. Additionally the layered architecture supports rapid prototyping, because the interface designer can change the presentation without having to rebuild the entire application.

The Seeheim model suffers from two main difficulties: Firstly, by replacing a presentation component, one usually had to rewrite the dialog to accommodate the new features of the new presentation component and secondly, since each dialog tended to need to be presentation-based, it requires to change the presentation component each time the dialog changes. Additionally, [Phanouriou, 2000] notes that the functional partitioning of the interface assumes a synchronous interaction between the user and the interface and does not support asynchronous modes of interaction such as direct manipulation where system feedback is interleaved with user's input.

3.2.3.2 Model View Controller Pattern

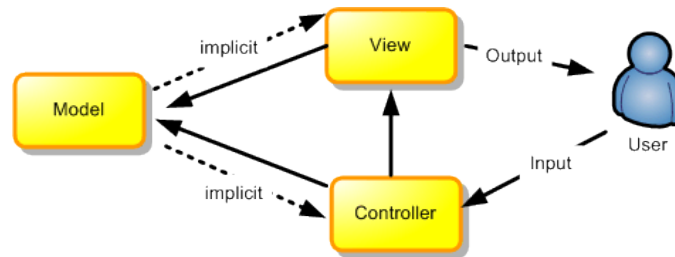


Figure 3.6: The Model-View-Controller concept.

The Model View Controller (MVC) paradigm divides the responsibility for a user interface into three components like illustrated in figure 3.6. A standard interaction cycle using the MVC paradigm is that the user makes some input that is received by the Controller component. The active Controller provides the interface between its associated Model and View and responds by invoking an appropriate action in the Model. The

Model represents the data structures of the application back-end and provides the source of the information to be presented by the user-interface. It carries out the requested operation, changes its state and broadcasts its change to all connected Views via the implicit link. After that each View updates its display by querying the Model.

The MVC concept is used in a lot of architectures as it provides a compelling object-oriented division at the abstract level, but concrete implementations often result in highly coupled model, view and controller classes. [Shan, 1990] notes that even experienced programmers require substantial learning effort before they can effectively use MVC. This is because the postulated component division is difficult to implement, since it often requires every model to be associated to a special view and controller pair. Assigning a different controller to a view does not change the interaction but often breaks the code and hinders the reuse of software components.

In MVC controllers are often involved in processing the semantics in addition to their defined role as an interface object. [Coutaz, 1987a] states, that in MVC the notion of control is distributed across all three components whereas it is centralized in PAC (see section 3.2.3.4). Since in most MVC implementations each MVC component includes the code to display itself, it is difficult to display it in more than one way or make global changes in the implementation. [B'far, 2004] criticizes the inherent asymmetry in treating the input and the output from the user to the model compounds the proliferation of controllers and view. Each new combination of interaction devices and their associated modalities require a new pair of a controller and a view component. Additionally each controller and its associated views must be synchronized to maintain consistency. The separation of input- and output capabilities of MVC is often impossible to realize, because input events are frequently “strongly related to immediate output feedback” [Coutaz, 1987a]. Further on this separation results in increasing message passing in direct manipulation systems where most system inputs alter the system states [Hussey and Carrington, 1996].

3.2.3.3 Arch-Slinky Model

An extension to the Seeheim model is the Arch-Slinky Model [Bass et al., 1992], which refines the level of abstraction. The Arch model expands upon the approach of [Nigay and Coutaz, 1991] that does not propose an architecture but focuses on the data exchange between the user interface and non-user interface portions of an interactive system.

The main goal of the arch model was tailored to minimize the future effects of changing technology by loosely coupling the remainder of a system from the interface toolkit allowing them to evolve independently from each other. As depicted in figure 3.7 the Arch model consist of five basic components: The Domain-Specific Component controls, manipulates and retrieves domain data from the application back-end. It communicates to the Domain-Adapter Component via Domain Objects that employ domain data end operations to provide functionality not directly associated with the user-interface. The Domain-Adapt Component mediates between the Dialogue and the Domain Specific Component. It implements Domain-related tasks that are required for the human operation of the system but are not available in the Domain Specific component. The

3 State Of The Art

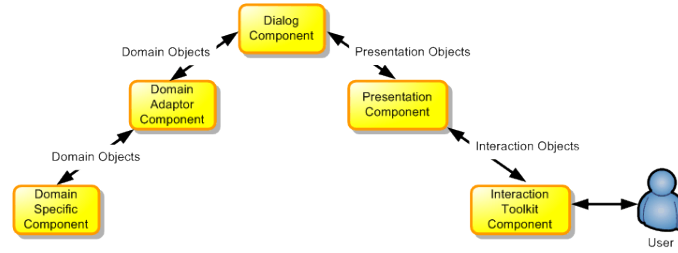


Figure 3.7: Arch Model.

Domain-Adapter Component instantiates different Domain Objects as the Domain Specific Component: They implement operations on the domain data related to the user interface. For example the Domain Adapter Component extracts useful elements of a data set of descriptions it has received from the Domain-Specific Component and filters data that isn't required to be presented on the user-interface.

The Dialog Component maps forth- and back between domain specific formalisms and user-interface-specific formalisms and has the responsibility for task-level sequencing for both the user and the application tasks that depend on user input. The Dialog Component communicates with the Presentation Component via Presentation Objects that control user-interactions by including data to be presented to the user and events to be generated by the user. Decisions about the representation of media objects are made in the Presentation Component that mediates to the Interaction Toolkit and provides a set of toolkit-independent objects (for instance a selector object that can be implemented as a radio-group or a menu). Finally, that Interaction Toolkit Component implements the physical interaction with the end-user via hardware and software. This component uses Interaction Objects that are specially designed instances of media-specific methods for interaction with the user to communicate with the Presentation Component [Bass et al., 1992].

The Slinky Meta-model for the Arch models The Slinky Meta-model for Arch was introduced to handle trade-offs such as minimizing the future effects of changing technology and for improving system run-time performance. The Slinky meta-model can be used to derive any number of architectures depending on specific developer goals and to evaluate architectures in terms of the desired goals.

The basic idea of the Slinky Meta-model is to allow a shifting of functionalities similar like the Slinky toy, that dynamically shift its mass when its in motion like illustrated in figure 3.8. Data-oriented systems with minimal dialog capabilities can instantiate an Arch Model with functionality concentrated in the Domain-specific and Domain-Adapter components, whereas dialog-oriented systems that have extensive capabilities for mapping user-actions and are oriented towards human-computer interaction can focus primarily on the dialogue part.

The Arch/Slinky Model is a major improvement over the Seeheim model, as it introduces a layer of abstraction of the user interface from the application back-end (via the

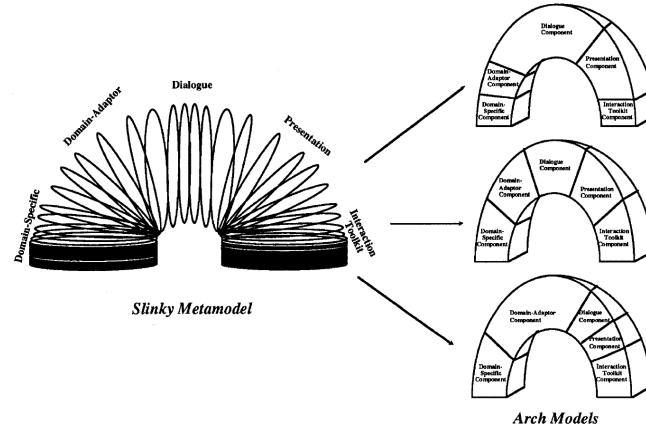


Figure 3.8: Derivation of the Arch Models from the Slinky Meta-model.
[Bass et al., 1992]

Domain Adapter Component) and the concrete user interface toolkit via the Presentation Component. The isolation between the toolkit and the back-end with the help of the presentation component enables reusing the user-interface on similar platforms (like Windows and Unix) with similar interaction capabilities. The support of platforms with different physical capabilities like for instance supporting voice interaction or devices with small screen require a re-design of the complete application.

3.2.3.4 PAC-Amodeus

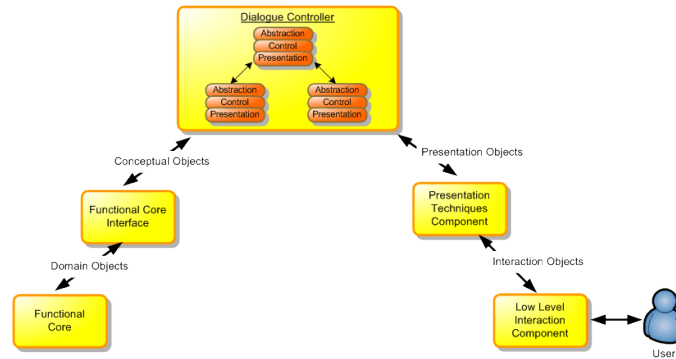


Figure 3.9: PAC-Amodeus Model.

The PAC-Amodeus model is a refinement of the Arch/Slinky Model and reuses it's main components (illustrated in figure 3.9): The Functional Core component implements the domain specific concepts and exchanges information with the Functional Core Interface (FCI) via Domain Objects, which is the same approach as designed by the Domain

3 State Of The Art

Specific component and Domain Adaptor Component in the Arch model respectively. The connection to the user is realized by the Low Level Interaction component that implements the basic interaction techniques such as a windowing system or a spoken dialog system. It communicates with the Presentation Techniques Component (PTC) (in Arch/Slinky: Presentation Component) via Interaction Objects in the same way like in the Arch/Slinky Model.

In contrast to the Arch/Slinky Model that does not define the organization of the dialog controller, PAC-Amodeus refines the component in terms of PAC Agents [Coutaz, 1987b]. The Dialogue Controller of PAC has the responsibility for doing the task-level sequencing and is decomposed into a set of cooperative PAC agents. These agents have to bridge the gap between the FCI and the PTC components of the PAC-Amodeus model regarding two aspects [Nigay and Coutaz, 1997a]:

1. Managing *data transformations* between the different formalisms used by FCI, which is driven by computational considerations of the functional core and the PTC component that reflects the toolkit/media capabilities of the user interface.
2. Handling of state changes in the FCI that require a *synchronization* of the associated PTCs (and vice versa) to achieve *consistency* between multiple views of a conceptual object.

The PAC agents of the Dialogue Controller form a hierarchy to reflect multiple levels for task sequencing, formalism transformation and data mapping. An abstraction process combines and transforms events coming from presentation objects to higher level events until the FCI is reached. Conversely, the concretization process decomposes and transforms high level data (Conceptual Objects) from the FCI into low level information. [Nigay and Coutaz, 1997a] introduce “facets” to express the three perspectives of a PAC agent: The *Abstract Facet* defines the competence of an agent to abstract and concretize events from the FCI, the *Control Facet* controls the dialog and maintains the mapping between the Abstract Facet and the Presentation Facet. Finally, the *Presentation Facet* implements the perceivable behavior of a PAC agent and is therefore related to some Presentation Objects.

The advantage of using a collection of cooperating agents for decomposing the Dialogue Controller is that each agent can be associated to a thread maintaining his own state to reflect the user’s activity. In this way each task or goal of the user corresponds to a thread of a dialog. This can be seen as an advantage of the PAC-Amodeus model over the MVC, that does not describe a dialog control. Different to MVC PAC-Amodeus groups input and output channel in one presentation component that is interacting with the user. Centralizing control for both channels in one component “reduces the total number of objects and the communication overhead between them” [Linton et al., 1989]. [Hussey and Carrington, 1996] mention that the structure of PAC provides easier maintenance than that provided by the MVC model but criticizes that the PAC model does not acknowledge the possibility of shared abstractions. Further on the hierarchy of abstractions of PAC can introduce redundant state information between low level interactors that are abstracted by associated high-level abstractions.

3.2.3.5 UIML Meta-Interface Model

The Meta-Interface Model is based on the Slinky Meta-Model of Arch and extends the level of abstraction. The model focuses on an abstraction to describe user interfaces for multiple devices. Therefore it refines the dialog part of the Slinky meta-model by separating it into four segments to describe a user interface: structure, style, content and behavior. The structure describes the organization of the parts of an user interface by a hierarchy and defines the different parts that are contained in the user interface. The style section specifies the presentation-specific properties of each part of the structure and enables to change properties like color, text, and fonts. The content section handles the information that should be presented by the user interface like for instance a list of items that should appear in a menu. Finally the behavior section describes the run-time interaction like events and application method calls by defining rules that are triggered when some condition is met (for instance if the user presses a button). The bundle of these four sections is called interface in the UIML Meta-Interface Model and describes the dialog between the user and the application (see figure 3.10).

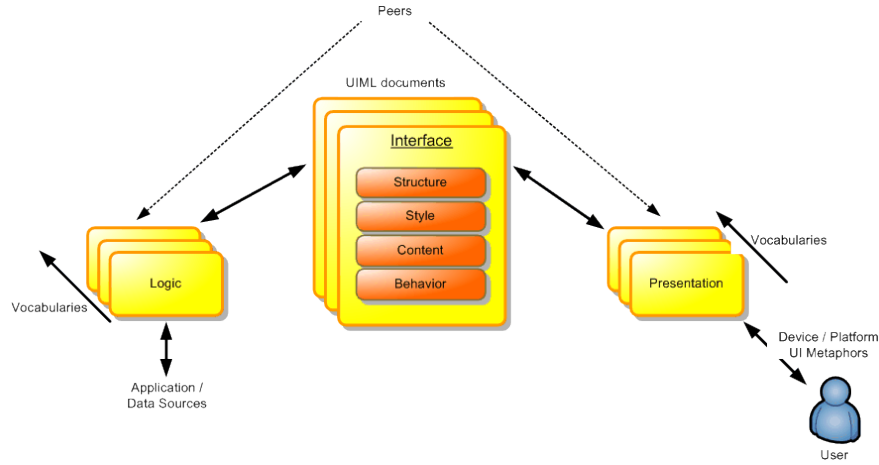


Figure 3.10: The Meta-Interface Model.

In the Meta-Interface Model both components, the Logic and the Presentation Component are described as peers that define mappings to external parts of the model. The Logic peers provide a canonical way for the user interface to communicate with an application while hiding information about the underlying mechanisms like data translations or method names for instance. The presentation peers describe a canonical way for the user interface to render itself while hiding information about widgets, their properties or the event-handling.

[Luyten et al., 2006b] mentions that the UIML Meta-Interface Model enables the reuse of large parts of the user interface across different platforms but criticizes that an abstraction of the user interface layout is missing, which limits the target set of devices that can be considered by the user interface rendering. Further on the missing layout abstraction can cause inconsistencies in the layout of the user interface. [Luyten et al., 2006b] pro-

poses an extension (DI-UML) that uses constraints and parametrized layout templates to raise consistency between the generated user interfaces.

3.2.3.6 Others

Beneath the architectural models presented in the previous sections a large number of agent-based models have been proposed along these lines. Similar to the PAC model, ALV [Hill, 1992], CNUCE [Paternó and Faconti, 1993] and York [Duke and Harrison, 1993, Duke et al., 1994] benefit from the agent-based style by stressing a highly parallel modular organization and distributing the state of the interaction among a collection of co-operating units.

[B'far, 2004] proposes an enhancement to the PAC agent model [Coutaz, 1987b] specifically targeted to serve mobile applications (PAC-TG). They insert two additional layers between the PAC control and presentation facets: A Generic Presentation Facet is restricted to encapsulate information and behavior about interactions with the user that are independent of the final user interface. The Generic Presentation Facet feeds a Transformation Facet that specializes the generic presentation objects for a specific end-device. Thus PAC-TG treats the concern of creating multiple user interfaces but does not consider the user feedback that might come from another device or modality in a multi-modal scenario.

3.2.4 Conclusions

Most of the architectural proposals to support interactive system development have been made between 1989 and 1994. Thereafter a continuous advancement of concrete patterns and framework proposals is missing to tackle the advances of modern, model-based interactive system development. Early approaches focused on describing patterns for the modularization of an interactive system to reduce the complexity of handling the event synchronization for graphical user interface that are able to present a whole bunch of interaction elements simultaneously to the user (PAC, ALV, CNUCE, York). Another important aspect of early approaches includes decoupling the functional backend from the user interface to force re-usability (MVC, Arch-Slinky).

Recent architectural advances like the UIML-Meta Interface Model or PAC-TG include enhancements to serve mobile applications and multi-platform user interfaces, but miss to describe an approach to consider the user interface models of model-based system development.

3.3 User Interface Description Languages

In the following sections an overview about the significant milestones of user interface description languages (UIDL) development will be provided. The presented UIDLs are grouped by the level of abstraction they offer: concrete description languages (section 3.3.1) have their focus on declaratively expressing a runnable user interface, whereas

model level description languages (section 3.3.2) introduce one or more (abstract) models that are involved in the development process of the user interface. Meta-model level description languages that will be discussed in section 3.3.3 add another layer of abstraction and describe how the models they contain are described.

3.3.1 Concrete Description Languages

3.3.1.1 XML User Interface Language (XUL)

Under the name XUL various approaches starting from Mozilla XUL to Microsoft's XAML have been proposed and most of them are organized in the Open XUL Alliance. The first approach for a XUL language has been done by the Mozilla Organisation to reduce the time-consuming development effort for building cross-platform software. Thus XUL approaches can be positioned to support cross-platform development like for instance Java but concentrate on user-interface portability. XUL started as an internal representation (the Gecko browser engine) of the Mozilla browser user interface.

XUL offers a separation of the application functionality, the presentation (realized as "skins", a combination of CSS² with associated images), and the language-specific text-labels. This approach allows altering the user interface presentation without changing the other parts. XUL has several "sister" languages: The eXtensible Binding Language (XBL) is used to declare the behavior of XUL widgets, the XPXOM / XPConnect Language part describes the language binding to access modules written in other languages like PERL, Ruby or Java, whereas XPConnect is a XPCOM module that is used to access C++ code from Javascript.

XUL has a strong focus on windows-based graphical user interfaces. Although the Mozilla organisation promotes XUL as a standard there exists a lot of derivations that focuses on special applications. For instance Mozillas XUL is not directly applicable to interfaces of small mobile devices, but Thinlets, a XUL derivat realizes a small sized XUL motor for this class of devices. Another XUL derivation is Microsoft's XAML. As XUL is an instance-level oriented decription language there are no abstractions of interaction functionalities supported.

3.3.1.2 XForms

XForms, a World Wide Web Consortium (W3C) Standard [Boyer, 2006] is an XML application targeted to form processing for the web. Beneath form-processing and form-validation, device-independence is one design goal of XForms. Therefore XForms is used to capture the intent behind a form processing application. It refines the XHTML form handling approach by specifying three parts for form processing: The XForms model, the user interface and the instance data.

XForms is focusing on graphical, interactive systems (web applications), for which it refines the HTML form processing by introducing form-validation mechanisms. Therefore

²Cascading Style Sheets, <http://www.w3.org/Style/CSS> last checked on 11/29/08

3 State Of The Art

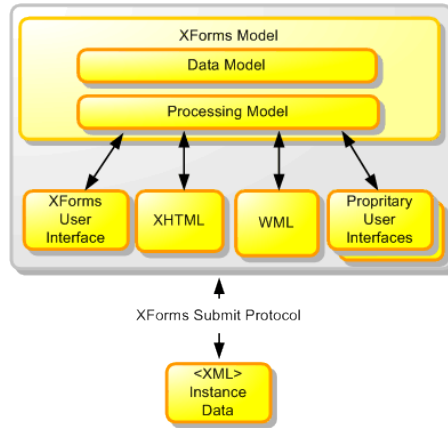


Figure 3.11: The XForms Language Model.

it requires an implementation for both, the (web-)server-side and the (web-)client. Client-sided data validation is used to avoid round-trips in data communication with the server side. The XForms specification mainly addresses the client where it defines the form-presentation, the event-handling and the validation.

As illustrated in figure 3.11 the XForms model can be subdivided into two basic parts: a data model and the processing model. The data model is used to structure the data that should be manipulated by the form later on. The XForms data model subsumes the XPath data model and therefore uses XPath expressions to reference the data. The term data model was chosen to reflect associations to the Model-View Controller based design of XForms, which like in MVC collects all essential data in the data model. The form controls of the XForms user interface reflect the View part of MVC. There is no specific component defined to reflect the MVC controller part, but portions of the processing model and the XForms events play a similar role [Dubinko, 2003].

The processing model defines the behaviors associated with the form. Thus, it defines what happens with the data, when a form gets submitted. Each user action is associated to form processing actions of the processing model. The second component of the XForm Language model is the XForms user interface, which represents the form-controls independent of a particular platform. Form-controls offer an abstraction by defining widget tags like input, text area, output, and submit, and can be rendered differently constrained by the capabilities of the target platform.

Finally, the instance data is used to exchange data between the server and the user and enables the possibility to collect the entered or manipulated data by the user (into the form) in an XML format back to the server.

The XForms language model can be considered as a description on an instance level as it primarily focuses on rendering and validating forms by offering a declarative description that partly follows the MVC model. Regarding the view part of the model, a basic abstraction for widgets in the form processing domain is offered, but mainly intended to be reified to a graphical-oriented user interface and supplement the current XHTML

form specification.

3.3.1.3 Others

The Webservice Experience Language (WSXL) promoted by IBM in 2002 [Chamberlain et al., 2002] focuses on two goals: Firstly it enables building web applications to a wide variety of channels and secondly it allows creating web applications from other ones. Similar to XForms WSXL separates presentation, data, and control components to ease the reassembly of multiple alternative versions [Souchon and Vanderdonckt, 2003]. WSXL specifies an adaptation description that can be associated with a WSXL base component to describe how the markup code generated by such a component can be adapted to new channels. WSXL is designed to be the next piece of the set of web services [Souchon and Vanderdonckt, 2003] and integrates well to other standards like XForms and Web Services for Remote Portlets (WSRP).

3.3.2 Model Level Description Languages

3.3.2.1 UIML

The User Interface Markup Language (UIML) is an interface meta language that is based on the Meta Interface Model introduced in section 3.2.3.5. It has been designed as a "vendor neutral, canonical representation of any user interface (UI) suitable for mapping to existing languages" [Abrams and Helms 2004]. UIML is separated into six parts that a user interface consists of: the structure, the presentation, the content, the behavior, the mapping to a user interface toolkit, and the binding to the application logic, which conforms to the Meta Interface Model. Different to the concrete user interface description languages that have been introduced in the previous section, UIML enables a designer to define or to integrate different user interface vocabularies. This has the advantage that once a UIML document that serves as a generic user interface description has been designed, this description can be mapped to any concrete user interface description language by using the mapping concept of UIML. UIML can be considered as one of the founding approaches that introduced a meta level to describe device independent user interfaces, but several major disadvantages have been identified over the years.

One disadvantage of UIML is that it defines a one-to-one mapping of abstract interaction objects into concrete interaction objects, which is not very flexible as [Luyten, 2004] mentioned. [Puerta, 2001] adds that UIML does not consider contextual data and is not intended to support knowledge-based system functions, nor does it target operation and evaluation functions. Further on, it does not clearly separate the rendering of the interface from the rest. More specifically, UIML relies on specifying at least one vocabulary and is not able to be rendered without one. [Abrams and Helms, 2002] has specified a generic vocabulary but it is constrained to graphical user interfaces and not modality-independent.

The Dialog and Interaction Specification Language (DISL) proposed by [Schäfer, 2007a] implements a platform and modality independent vocabulary and extends the behavioural part of UIML to allow rendering the user interface by a DISL

renderer that directly runs on the end devices. These renderers directly interpret the DISL language, which defines modality-independent generic widgets.

3.3.2.2 AUIML

The Abstract User Interface Markup Language is an approach by IBM [Azevedo et al., 2000] to specify abstract user interfaces to overcome that explosion of transformations that are required to map from various source markup languages to the continuously growing concrete user interface markup languages. Therefore AUIML offers keeping the “intent” of the user interface separate from its rendering. But since AUIML is mostly concerned abstracting the user interface logic, a behaviour model has not been specified to our knowledge, which prevents allowing a modality independent specification of the user interface and in fact the currently available toolkit is only able to produce HTML and Swing user interfaces as [Schäfer, 2007a] notes.

3.3.2.3 TERESA XML

TEREASA XML defines a set of XML-based model descriptions for the TERESA authoring environment (Transformation Environment for inteRactivE Systems representations) [Mori et al., 2004]. TERESA XML is composed of two main models: a task modeling language, which is an XML-based representation of the CTT notation and an abstract user interface model. Additionally exemplary instances for concrete user interfaces, such as XHTML-based graphical user interfaces or a voice user interfaces are supported [Berti et al., 2004]. Whereas the abstract user interface model part is used to define abstract interaction objects (AIO) during the design process, the task model part is refined to a set of connectors specifying the order of processing the AIOs. Therefore the design results in a state machine defining the raw dialog of the application.

3.3.2.4 Seescoa XML

SEESCOA XML [Luyten and Coninx, 2001, Luyten et al., 2002] has been developed as a user interface description language to describe abstract interaction objects as part of the Dygimes System [Luyten, 2004]. The language focuses on describing the presentation level on mobile devices requiring only a small footprint for rendering user interfaces. Therefore the set of supported interactors is very close to HTML and only basic form-based applications can be designed. A behavior model is not addressed by the language, but a basic eventing system that can be included into the language renderer by implementing “action plugins” allows an integration of custom communication protocols. Since SEESCOA XML has been originally designed as a “serialization” language, it is in syntax close to program objects and can be efficiently interpreted by a machine.

3.3.2.5 UsiXML

The USeRInterface eXtensible Markup Language (UsiXML) is aimed at offering a comprehensive approach for describing the various levels of details and abstractions of a user

3 State Of The Art

interface depending on the context-of-use. Therefore it is structured following the basic abstraction levels offered by the Cameleon Reference Framework that has been previously discussed in section 3.1.3.4.

At the task & concepts level of the Unified Reference Framework UsiXML [Limbourg et al., 2004b, Limbourg et al., 2004a, Limbourg, 2004] offers a *task model* that takes advantage of Paterno's CTT notation, and a domain model that describes the objects and classes that are presented and manipulated by the user. At the abstract level, UsiXML's *abstract user interface model* specifies the groups of tasks and domain concepts that will be presented together at the same time to the user, whereas at the concrete level the *concrete user interface model* is used to give a detailed specification of the user interface appearance, that is dependent on a certain modality (for instance to generate a graphical or a voice-based user interface).

Different to earlier approaches, that mainly focus on declarative descriptions of models on certain abstraction levels (like UIML or XUL) UsiXML explicitly considers two additional aspects: the context-of-use and support for a transformatal development approach.

The context-of-use is considered by specifying three additional models: the *user model* that decomposes the users of the interactive system into stereotypes and describes them by attributes for instance to express their experience with the system or a specific task or their motivation. The *environment model* describes the global environment where the interaction between the user and the system takes place. Properties of the environment can be physically to describe the level of noise or the lightning, or psychologically to express the level of stress of the user. Finally the *platform model* is used to specify the relevant attributes of the hardware and software of the device that is used to present the user interface.

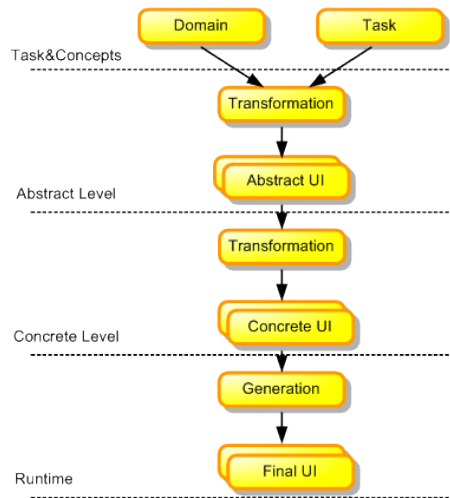


Figure 3.12: Transformation Approach in UsiXML.

The transformatal development approach support of UsiXML is motivated by the abstractions of the Cameleon Reference Framework and is based on graph transformations

[Puerta and Eisenstein, 2003] between the models of the different abstraction levels (illustrated in figure 3.12). The transformation system of UsiXML is organized by graph rewriting rules equipped with negative application conditions and attribute conditions. This transformation can be interactively constructed or manipulated using an Editor [Stanciulescu et al., 2005] or can be processed automatically. Up to now, the transformations are used between the task&concepts and the abstract user interface level, and between the abstract and the concrete user interface level, respectively.

3.3.2.6 GIML

The Generalized Interface Markup Language (GIML) [Kost, 2006] supports four models of abstraction similar to the UIML approach: structure, style, content and behavior that compose a GIML document. This document is mapped to a specific user interface by using an XSLT-based mapping or a renderer that interprets the GIML language for a specific target platform. As the dialog model includes navigation elements, GIML supports rendering user interfaces to three dimensional presentations offering a lot of navigational options and voice prompts (which requires a strong navigational sequencing) as well. GIML does not consider a user-centric design, nor includes a task model, but instead follows a functional and technological-driven approach.

3.3.2.7 XISL

The eXtensible Interaction Scenario Language (XISL) [Katsurada et al., 2003] has been specified to describe multi-modal user interfaces. It basically consists of an abstract dialog flow description that is able to synchronize dialog flows between several connected modalities similar to SALT and XHTML+Voice. Different to these approaches XISL “abstracts” from specific modalities, by distinguishing between input and output modalities that can be addressed in parallel, alternatively or sequentially. Each modality and platform that should be addressed by XISL needs to be implemented separately, since XISL does not include a transformational user interface development approach.

3.3.2.8 RIML

The main design goal of the Renderer Independent Markup Language (RIML) is an automated pagination support [Ziegert et al., 2004, Spriestersbach et al., 2003]. RIML mainly targets to form-based web applications as it defines a XHTML 2.0 language profile, which can be enhanced by a speech-based access. Therefore it separates the content definition from the description of the dynamic adaptation, which can be performed by a user interface. The layout mechanism of RIML is simple but effective. By supporting the definition of rows, columns, and grids it is similar to AUIML but more powerful as RIML supports specifying alternative contents for the different output channels and an intelligent pagination supporting to fit user interfaces to different screen sizes. [Luyten, 2004] criticizes, that RIML merges parts of XHTML, XForms and other markup languages resulting in a fairly big specification.

3.3.2.9 Layouting Models

The user interface layout specification has not been explicitly addressed in the model-based user interface description that have been presented in the last sections. Nevertheless initial ideas and concepts relevant for the layout design and specification should be discussed in the next paragraphs.

Nichols et al. list in PUC [Nichols et al., 2002] a set of requirements that need to be addressed in order to generate high-quality user interfaces. As for layout information they propose to not include specific layout information into the models as this (a) tempts the designers to include too many details into the specification for each considered platform, (b) delimits the user interface consistency and (c) might lower the chance of compatibility to future platforms. Even though we share the idea to use a hierarchical grouping to express the user interface organization; we are looking for a more flexible approach that does not require a hierarchy as a key element.

The SUPPLE system [Gajos and Weld, 2004] treats interface adaptation as an optimization problem. Therefore SUPPLE focuses on minimizing the user effort when controlling the user interface by relying on user traces to estimate the effort and to position the user interface widgets on the user interface. Although they present an efficient algorithm to adapt the user interface it remains questionable if reliable user traces can be generated or estimated if a user uses the same application for various tasks. While SUPPLE also uses constraints to describe device and interactor capabilities they present no details about the expressiveness of the constraints and the designers effort in specifying these constraints. The layout of user interfaces can be described as a linear problem, which can be solved using a constraint solver.

Recent research has been done by Vermeulen [Luyten et al., 2006a] implementing the Cassowary algorithm [Badros et al., 2001], a weak constraint satisfaction algorithm to support user interface adaptation at run-time to different devices. While he demonstrates that constraint satisfaction can be done at run-time, to our knowledge he did not focus on automatic constraint generation. Other approaches describe the user interface layout as a space usage optimization problem [Hosobe, 2001], and use geometric constraint solvers, which try to minimize the unused space. Compared to linear constraint solving, geometric constraint solvers require plenty of iterations to solve such a space optimization problem. Beneath performance issues an efficient area usage optimization requires a flexible orientation of the user interface elements, which critically affects the user interface consistency.

Richter [Richter, 2006] has proposed several criteria that need to be maintained when re-layouting a user interface. Machine learning mechanisms can be used to further optimize the layout by eliciting the user's preferences [Gajos and Weld, 2005]. The Interface Designer and Evaluator (AIDE) [Sears, 1995] and Gadget [Fogarty and Hudson, 2004] are incorporating metrics in the user interface design process to evaluate a user interface design. Both projects focus on criticizing already existing user interface layouts by advising and interactively supporting the designer during the layout optimization process. They follow a descriptive approach by re-evaluating already existing systems with the help of metrics.

3.3.2.10 Others

Beneath the user interface description languages supporting the definition of the user interface on an abstract, model-level that have been presented in the previous sections several others have been proposed over the years. AAIML [Zimmermann et al., 2002b, Zimmermann et al., 2002a] supports an abstract definition of interactors, a classification of interactors by feature classes to ease the identification what interactor can be rendered for which devices, and attaching help texts to individual interactors. TADEUS XML [Mueller et al., 2001] offers a more comprehensive approach by also referring to a device description and including a basic mapping approach. They abstract concrete user interfaces by the definition of abstract user interface objects that can be transformed to specific user interfaces by considering the target device description in HTML, WML and Java-AWT. The Reusable Device independent Interaction XML (RDIXML) [Martikainen et al., 2002] provides further models within one representation, namely task, domain, user, dialog, presentation, device, and application models. The task model of RDIXML is the most advanced one. It's on the one hand the starting point of user interface development and on the other hand, it relates all mappings to the task model as the central model. As [Schäfer, 2007a] notes, the focus of RDIXML lies on reusability and therefore other aspects such as modeling relations between tasks are not well developed. Further on, some models are not cleanly separated, thus for instance the presentation model includes the dialog model.

3.3.3 XIML Meta Model-level Description Language

Different to model-level description languages described in the last section, meta model-level description languages rise the abstraction to a level that enables the description of the models the languages utilizes. A description on the meta-model level can offer an extension-mechanism to add or refine the models of the language.

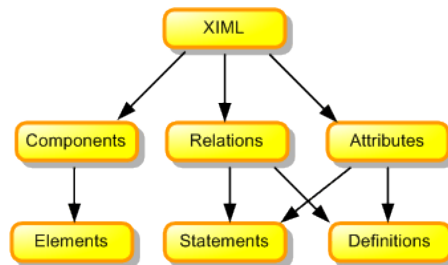


Figure 3.13: Basic representation structure of XIML.

The eXtensible user Interface Markup Language (XIML) was one of the first model-level user interface description languages that included an extension mechanism by offering an abstraction on a meta-model level that organizes the representation structure of the XIML language as depicted in figure 3.13.

There are three main representation units defined within XIML: *interface components*, *relations*, and *attributes*. Interface components are used to categorize a collection of

interface elements. Each element can belong to more than one interface component. As new components can be defined to extend the XIML language and no limit on the amount of new components and their associated elements has been set, XIML 1.0 proposes five basic interface components, three on an abstract level (task, domain and user component) and two on an implementational and concrete level (dialog and presentation). *Relations* can be defined to specify links between interface elements that are associated in the same or in different interface components. [Puerta and Eisenstein, 2001] note that an explicit representation “creates a body of knowledge that can support design, operation, and evaluation functions”. XIML distinguishes between relation definitions and relation statements. The former ones specify a canonical form of relations whereas the latter ones specify actual instances of a relation. Finally *attributes* are defined as features of properties of elements in XIML that can be assigned a value. Beneath supporting basic data types, instances of other existing elements can be set as an attribute value.

3.4 Conclusions

The state of the art overview revealed the long history of model-based interactive system development. We inspected three aspects of model-based development: The method, the development of architectures, and the most prominent user interface description languages and notations supporting such a development approach. As there are a lot of different approaches that have been proposed especially for methods and languages in the last two decades, we structured the analyzed proposals for development methods corresponding to the time they have been proposed in and figured out four important generations (milestones) to characterize methods and tools. Each generation is motivated by new development requirements and the availability of new technological possibilities.

When inspecting the various user interface description languages available, we have identified three different layers of abstraction that languages could be characterized. Concrete description languages have their focus on enabling a declarative description of user interfaces. Model-level description languages describe the user interface through a set of declarative models, each on an different level of abstraction or by specifying a certain aspect of the interactive application. Finally, meta-level languages support the specifications of the various models that they are able to handle and therefore offer the definition of comprehensive and consistent sets of models. Different to related work on languages and methods that have been continuously advanced by the research community, the development of architectures and paradigms has not been significantly advanced after an early development phase between 1987-1997. Thus, the progress in model-based systems in terms of methodology and languages can not be mapped to an architectural pattern. Since early architectural proposals mainly focus on identifying a separation of concerns and support modularity by hierarchy, they fail to address the fact of supporting model instantiation on all levels of abstraction but require a model compilation to a final user interface to be executable.

The state of the art analysis motivates our approach to be realized in a three-folded way. Whereas the development of a tool-supported and comprehensive method is the

3 State Of The Art

most important aspect of our work, we complement our approach by an proposal for an agent-based architecture supporting the direct model execution to remove the actual gap between design and run-time of an application. Finally, we do not reinvent the wheel by proposing yet another user interface language, but propose language additions where we experienced missing aspects, which we indentified during the implementation of several prototypes and case studies. The next three chapters address our three-folded approach by starting with the derivation of the requirements first and thereafter presenting our approach to tackle down the identified problems regarding the method (chapter 4), the architecture (chapter 5) and the language (chapter 6).

4 The MASP Methodology

User interface design is often considered an art and requires human intervention for all the creative processes like specifying the layout, selecting suitable colours and widgets for instance. Opposite to this is a business perspective that demands for deadlines, a calculation of efforts required to implement an interactive system, and measurements that can give a basic indication of the project progress. Another counteracting axis can be identified by the evolving technologies offering new kinds of interaction principles by the continuously advancing device and computer capabilities. Even small changes like updates of the underlying operating system or a new display form factor can confront a user interface developer with a big problem. End users expect applications to be integrated seamlessly into their devices, supporting all interaction capabilities and usage paradigms as well as the realization of homogenous usage principles enabling switching between the device's applications without the need for learning about the position and working modes of the basic application controls.

This is the actual area of conflict where engineering methods can help to guide software developers to handle conflicting requirements along the three conflicting axis and tools can help to automate reoccurring tasks that do not require the humans' creativity. In the following sections a method for interactive system development will be proposed that offers a clear separation of concerns, a continuous tool support and the option to integrate the targeted audience (the end user) in all steps of development process to incorporate feedback as early and often as it makes sense to the involved parties.

The ultimate goal of a model-based development of user interface is to minimize the implementation phase by the systematic use of "productive" models that can be processed by a machine. This requires an engineering process that is described with explicit models, which are linked together through mappings and transformations. The process focuses on the analysis and design phases, where all the models are designed by applying transformations that guide to lower level models. The subsequent implementation phase mainly has to focus on selecting and configuring a framework that is able to execute these models and will be only shortly described to complete the process.

4.1 Requirements Derivation

In this section we systematically derive the requirements by presenting the observations and shortcomings that we figured out during the state of the art analyses of related work in user interface development, which has been presented in section 3.1. From the list of observations we then derive a set of requirements to support our concept for a tool-supported method for interactive system development.

4.1.1 Observations

We can observe three aspects when looking at related work of interactive system development methodologies. First, we identified a diversity of methods. Second, we experienced a heterogeneity in phase coverage between the various approaches and finally, we observed that several tools have been proposed in the past to support the designer and user during development so far. In the next paragraphs we detail these observations.

4.1.1.1 Methodological diversity

On the one end of the spectrum, processes have been proposed that put a strong focus on the specification phase. They start by investigating in the domain of discourse by analyzing and designing the relationships between the involved entities. On the other end of the spectrum, processes start by designing a user interface with the help of a user interface construction tool enabling the design of the final presentation first, before the underlying processes and functionalities are investigated. The former processes implement a forward engineering whereas the latter implement a reverse engineering approach.

Up to now there does not exist a general agreement regarding the order of the process steps in the analysis phase. Some approaches, like [Ziegert et al., 2004] start by identifying objects first, whereas others propose to start by a task analysis (for example [Paternò, 1999]) before identifying the domain objects. Some approaches like the Cameleon Reference Framework [Calvary et al., 2003] leave it to the designer to decide upon the initial step. UsiXML [Limbourg et al., 2004a] and the unifying reference framework support different entry points to the user interface development process that can then be continued by both, forward and backward engineering.

4.1.1.2 Heterogeneity in phase coverage

One reason for the actual diversity of methods that has been proposed to support interactive system development is about the heterogeneity in phase coverage. Most approaches focus on the analysis and requirements phase (like [Paternò and Mancini, 1999]) or concentrate on the design phase of the development process [Calvary et al., 2003, Mori et al., 2003], on dialogue modeling [Vanderdonckt et al., 2000, Reichard et al., 2004, Dittmar and Forbrig, 2004] or on designing context-of-use adaptations [Luyten, 2004, Clerckx et al., 2007] but do not consider a comprehensive tool-supported method. Further work has been undertaken, which focuses on the identification of mappings between various design models [Stanciulescu et al., 2005, Clerckx et al., 2004b] or notations like for instance UsiXML have been proposed that can be utilized as a standardized exchange format to connect the phases of the development process [Limbourg et al., 2004b, Limbourg and Vanderdonckt, 2004b].

4.1.1.3 Tool-support and user involvement

As tool-support is an widely accepted aspect of model-based user interface development, various tools have been developed to support the development process. The

task analysis is supported by tools like for instance the “Eliciting Task Models Tool” [Paternò and Mancini, 1999], which helps the designer to derive task structures from scenario descriptions. The Dygimes Framework [Coninx et al., 2003], an extended version of Seescoa [Luyten et al., 2003c], starts by a task modeling phase using the ConcurTask-Trees notation to derive an abstract user interface and to transform it to the final user interfaces for different platforms.

Whereas task analysis and modeling is done on a very abstract level to design interactive systems, prototyping tools like for instance SketchiXML [Coyette et al., 2004, Coyette and Limbourg, 2006] support Low Fidelity Prototyping by processing hand drawn user interface sketches and automatically transforming these sketches to graphical user interfaces. A different approach is followed by high fidelity prototyping tools (like for instance GrafiXML [Florins et al., 2006b], VisiXML [Sluys, 2004] or LiquidUI¹ from Harmonia Inc.) that enable computer-aided design of user interfaces by offering pallets of widgets that can be used for user interface prototyping and can be layouted by using drag-and-drop metaphors.

Both, tools supporting task analysis and modeling as well as prototyping tools facilitate user involvement in the user interface design process. Task models can be designed by a tool like the CTT editor [Mori et al., 2003] and scenario-completeness can be verified in that way in cooperation with the targeted audience of the interactive system. Prototyping processes often focus on realizing mockups of user interfaces as early as possible to enable a quick preview of the user interface experience for the end-user in order to consider feedback as early as possible.

4.1.2 Shortcomings

Regarding the methodological observations two basic shortcomings can be identified. On the one hand, the lack of continous tool support currently is an aspect that prevents a greater acceptance of model-based system development. On the other hand, most approaches focus on designer tools but do not consider testing support. Both shortcomings will be described in the next sections.

4.1.2.1 Lack of continous tool support

Tool support is limited to support the developer or designer in selected phases following a model-based approach. The UsiXML user interface description language has been proposed as a standard exchange format between various tools for prototyping, reverse engineering or simulation of user interfaces. Up to now the tool support is limited to the user interface specification at design-time. The Teresa tool offers an integrated development environment that starts with task analysis followed by task modeling. After that an abstract user interface can be derived and refined for various platforms to end up in a final user interface. Similar to the UsiXML tools the run-time support is limited and the adaption to different platforms is mainly done at design-time. The Dygimes framework offers a run-time environment for the provision of user interfaces for various

¹<http://www.harmonia.com>, last checked 07/10/08

platforms and sets a strong focus on run-time adaptation of user interfaces, for instance by a constraint-based layouting mechanism that is able to relayout existing user interfaces based on templates regarding the capabilities of the targeted end-devices. Tool support of Dygimes is limited to support task analysis and modeling based on the CTT editor of Paterno et al. In UIML there are GUI builders available that can support the designer in the user interface creation process, whereas XIIML is supported by a tool that allows simulations considering temporal relations between task and design decisions for the navigation dialogue [Forbrig et al., 2004].

4.1.2.2 Lack of continuous testing support

Beneath the participation of usability experts, testing support in the development process can have significant advantages for the final quality of the interactive system. Well-known software engineering models like the improved V-Model for instance consider testing steps associated with each phase of the anticipated development cycle. Recent approaches that implement agile development claim for close involvement of the targeted audience. Regarding development of interactive systems prototyping and task-modeling are widely accepted approaches that support user-involvement. The former one actually gained a lot of attention in the model-based research community by Vanderdonckt et al. that propose various tools to support prototyping on different levels of fidelity. The latter one got strong support by Paterno et al. proposing various tools for task analysis and task modeling based on the CTT notation. Some kind of continuous testing support is still missing as actual approaches consider testing the user interface in certain aspects only. Although task modeling notations like for instance CTT implement a comprehensive syntax and easy accessible semantic the end-user often prefers to talk in form of scenarios [Paternò and Mancini, 1999] and leaves the task modeling up to the designer. The CTT tool allows to prove the modeled task-trees by enabling the user to simulate the designed interactive application before the interactive system is finally implemented, whereas other approaches like IdealXML or [Reichard et al., 2004] give an abstract view on a visualized prototype illustrating the basic user-interface structure that has been automatically derived from a task tree.

4.1.3 Methodological Requirements

By identifying the actual shortcomings of methods to support interactive system development, five requirements can be derived that should be tackled down by our work and will be presented in the following sections.

4.1.3.1 Lifecycle coverage

The proposed methodology should cover the whole life-cycle of an interactive application. Following a model-based approach, the methodology should cover all states of software engineering starting from early requirements analysis to design, prototyping, implementation, deployment, maintenance and adaptation at run-time. Since an application of the proposed methodology should guide the involved roles to an interactive system that can

be accessed via various platforms, each step in the process has to be well defined and the relevant information to be considered should be explicitly stated in order to complete a step of the methodology.

4.1.3.2 Automated process with support for interactive development

A computer assisted approach for model-based user interface generation should provide the participating roles the possibilities to automate repeating tasks while offering ongoing control. The often mentioned critique of model-based approaches, which is about the difficulties for the involved roles to understand how the model manipulations affect the resulting user interfaces, requires an explicit involvement of the affiliated roles. This demands for an interactive design process enabling comprehensive model manipulations on all levels of abstractions.

4.1.3.3 Automation of repeating tasks

Repeating tasks should be stored in form of a design knowledge base, which should gather re-appearing work in form of templates or macros. Thus a high-level automation is required that is able to off-load routine. The methodology should support the decomposition of high-level tasks into simpler ones and has to keep track of the status of the design activities.

4.1.3.4 Support for multi-path development

The methodology requires to promote some form of a “guided path” through the user interface development process. The path has to include early requirements definition, the conceptionalization and implementation. As proposed by [Luo, 1994] in the humanoid approach, development following the methodology must be flexible to be easily adapted to different forms of organisation. This requires flexible development approaches (top-down, bottom-up, wide-spreading, and middle-out) and additionally enables the involved roles to flexibly shift between these approaches. Limbourg [Limbourg et al., 2004b] extends this requirement and formulates a demand for “development steps (that) can be combined together to form a development path that is compatible with the organisation’s constraints, conventions, and context-of use” [Limbourg et al., 2004b].

4.1.3.5 Support for explicit involvement of the targeted audience

Supporting continuous involvement of the target audience can improve the quality of the interactive applications measured against what the targeted user-group expects. Similar to already established software engineering models, like the enhanced V-Model that considers tests for each phase of the development process, usability-tests and acceptance-tests associated to every step of the methodology can help to catch potential shortcomings. Intermediate results might be refined by considering user’s feedback and explicit goal-and-process management is required to prevent try and error trashing as Carroll notes in [Carroll, 1992].

4.2 The MASP User Interface Development Process

Depending on the type of the interactive system that has to be developed, user interface development processes can look very different. For example processes for interactive systems that are specially designed to run on a desktop often have a strong focus on user interface development using GUI builders, allowing to construct user interface mockups relying on the platform specific widgets offered for example by Microsoft Windows or Java Swing. Results of the (process) analysis are used for a basic dialog design of screens, windows and messages to the user.

Different to user interface development for desktop applications, the development of web applications is not limited to utilize a special widget set, because interaction design using the basic form elements of XHTML is restricted to form design. Thus, web application design often considers picture-based mockups, already including a corporate design and specialized widgets, like horizontal or vertical menu bars and a proprietary dialog navigation not conforming to the browser back-and-forward navigation principles. All of these approaches have in common that they are able to realize prototypes quickly and can test the user's requirements early.

Model-based approaches are targeted to multi-platform user interface development and therefore offer principles of abstraction via the usage of models to broaden the view of the target platforms that can be considered as the final user interface. As the level of abstraction has been continuously increased by introducing various models that bridge the gap between abstract modeling of concepts and tasks and the final user interface design, the temporary distance between the design of the abstract models until the first prototype reflecting the final user interface has been increased simultaneously. To deal with the increasing distance between analysis and previewing first results, a method is required that enables a set of testing steps for each phase of the process. Different to traditional software engineering processes that spend a lot of attention to functional testing by using unit testing for instance, a user interface design process interprets testing as involvement of an additional (external) role like a domain expert, the targeted audience of the application or usability specialists.

In the following sections of this chapter our method for model-based interactive system development will be presented. The method covers the analysis, design, and implementation phase of a software engineering approach and is complemented by a run-time architecture that enables the direct deployment of the results of the design and implementation phase in a run-time system. The overall goal of a method for interactive system design is to replace ill-structured and add-hoc approaches for user interface development with a structured and user-centric process that guides the user interface developer from early requirements analysis to a final running interactive system. Compared to ad-hoc interactive system development by pictures or GUI mockups that are designed decoupled from system design and often after the systems functionalities have been developed, a structured model based process has several advantages:

- The development process is transparent for all participating actors. At any time of the development process it can be clearly stated, which steps have been done so

far and more important: what parts and costs remain to finish the system.

- The process starts by analysing the requirements of the user for both, the system's functionality and its expected interaction behavior. This prevents the development from undesired functional features without considering how they can be efficiently controlled and accessed by the user.
- By supporting an abstract-to-concrete modeling the design of large and complex systems can be developed. Abstraction is a human ability that is generally utilized to understand complex problems and big systems. The utilization of abstract design models in the process ensures that there is at least one level of abstraction to describe the complete system in one model enabling to survey the overall system design. Since all of the models are connected or derived from each other, even modeling details on a lower abstraction can be done with regard to the other parts of the system.
- The development of multi-platform, multi-context and multi-modal interactive systems ends up with an explosion of user interfaces for a single application that can only be reduced by introducing abstract design models that capture general aspects of the user interface to be considered on all platforms but are realized differently for each platform.
- User interface consistency is an important aspect that needs to be maintained for multi-platform, multi-context and multi-modal interactive systems to remain usable after a context-change has been occurred. By a process that derives abstract models to more concrete ones, the designer can be guided to maintain the consistency.

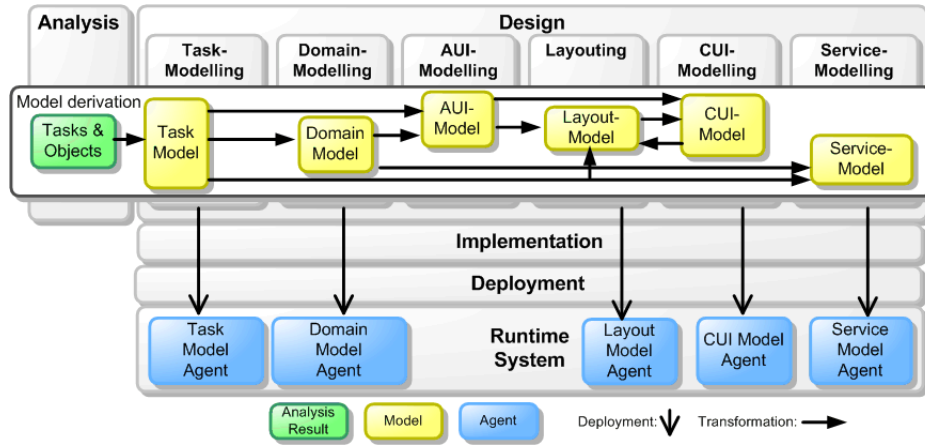


Figure 4.1: Basic structure of the development process creating run-time models and transient models.

Figure 4.1 illustrates our development process that comprises five phases: analysis, design, implementation, deployment and maintenance at run-time. Compared to other

4 The MASP Methodology

approaches for model-based system development our method offers the following benefits:

- The complete coverage of all phases of a software engineering process by a model-based interactive system development method.
- The gap between design and implementation of an interactive system is bridged, since most of the design models are designed to be directly deployed, interpreted and maintained as part of a user interface runtime system.
- Continuous tool support in all phases of the development process to ease the specification of the design models by interactive tools.
- Continuous testing support in all phases of the process to reduce the risk of misunderstandings and misconceptualizations as early as possible and keep design iteration cycles short.
- The method introduces a new model for specifying the layout of a graphical user interface that has not been considered by other approaches and additionally proposes a different abstract user interface model based on explicitly considering domain model information.

The design and development of our models is done by using a transformational approach that is illustrated in figure 4.1. Each horizontal arrow depicts a model-to-model transformation. In the following description of the method we focus on forward-engineering and only describe unidirectional transformations from abstract to more concrete models or bidirectional transformations between models of the same level of abstraction. But reverse-engineering can be supported as well by introducing model-to-model transformations that work in the opposite direction and are transforming low-level models to high-level models. The vertical arrows illustrate that most of the design models can be deployed to get directly interpreted into our run-time environment. Therefore an interactive application consists of a set of model agents, each specialized to execute one of the design models. On the one hand the direct deployment bridges the gap between design and implementation and reduces the additional introduction of low level source code to a minimum. On the other hand it enables instant testing of the design models since the model agents can be executed independently from each other and each agent supports a visualization that can be used for end-user testing.

In the recent years tools have been proposed, most supporting certain phases of development. Especially task-based analysis and design are widely accepted as a suitable abstraction level and starting point for a user-centered method within the model-based user interface research community [Vanderdonckt and Berquin, 1999, Paternò, 2005].

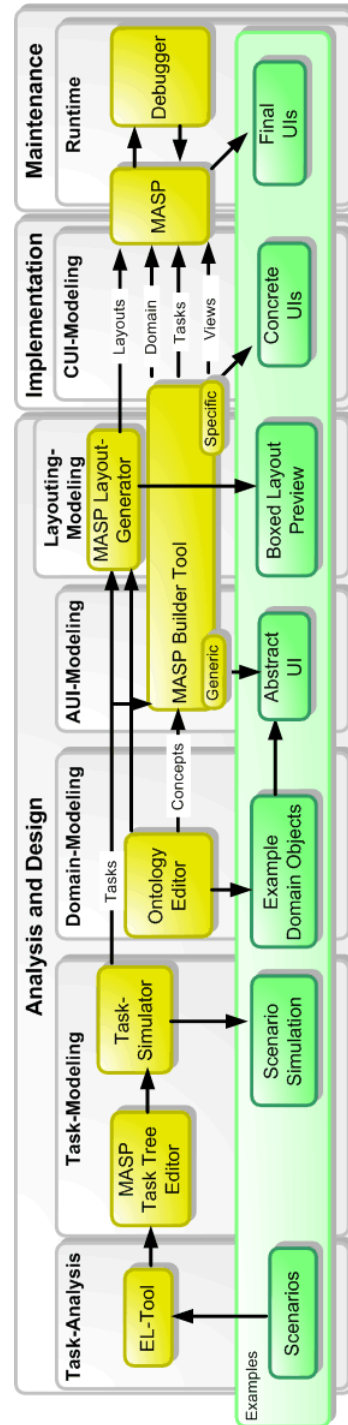


Figure 4.2: All phases and steps of the MASP methodology are supported by interactive tools. Additionally continuous testing is possible in all steps of the process by deriving exemplary objects, task or user interface simulations and model checking.

Figure 4.2 illustrates the sequence of tools that support the method and will be introduced in the following sections. Whereas in the early analysis phase and the domain modeling we rely on already existing tools (the EL-Tool [Paternò and Mancini, 1999] and the JIAC ontology builder [Tuguldur et al., 2008] respectively) we introduce several new tools to support the designer in all other steps of the method. The method has a strong focus on testing supports to uncover misunderstandings and misinterpretations as early and often as possible. Therefore all tools support generating previews (illustrated by the solid-bordered boxes in the bottom layer of figure 4.2), enable model checking or offer to run simulations enabling testing in all phases of the development process.

In the following we describe all phases of the method in detail. Each phase consists of sub-phases. Thus for instance the design phase is concerned with task modeling, domain modeling, abstract user interface (AUI) modeling, layout modeling, and concrete user interface design. A sub-phase can be subdivided into a set of subsequent steps that enable the manipulation of the models. Each step produces an intermediate result that serves as the input for the next step. Like the models, this intermediate result defines a state that can be saved and continued later on or even shared within the development team. We distinguish between different types of steps: *Automatic steps* identify steps describing activities that can be automated and do not require human intervention, whereas *interactive steps* specify steps that require decisions by a human. *Testing steps* enable the production of intermediary results that can be used to test the actual state or a part of the interactive application to be developed. Such testing steps produce for instance a prototype enabling the simulation of certain parts of the functionality or the final look-and-feel of the interactive system that has to be developed. Unlike functional testing that often gives some sort of positive or negative feedback or some kind of a measurement referring to a metric, in our method testing is done by an interactive process and ends up with an improved prototype.

The next sections describe all the steps of all the sub-phases in greater detail. For a better understanding each sub-phase will be illustrated by a schematic figure. There we use boxes with a dotted border to illustrate results of a successful development step. Solid lined boxes notate interactive processes requiring the developer, which can be supported by a tool and orange boxes identify testing steps to test the results of a development step. Arrows specify the sequence of steps that produce the results. The link relationship describes results that are referencing each other, different to nested boxes, which illustrate result composition.

4.2.1 Analysis Phase

We consider scenarios to provide an informal initial input to the development process to derive the requirements for the interactive system. They are written together with involvement of the user of the interactive system. Scenarios are stories about people and their activities [Carroll, 1999] and include a *starting state*, a *setting* [Potts, 1995] and at least one *actor* (to answer the question: "who is the story about?") that owns a *role* in the scenario. Further on each scenario has to contain at least one *goal* for an involved *actor* that will be achieved in the scenario and therefore is considered as the defining

goal for a scenario and answers the question “why did this story happen?”. The plot of a scenario follows a set of activities that the actors are doing and events that happened to them in order to reach the defined goal.

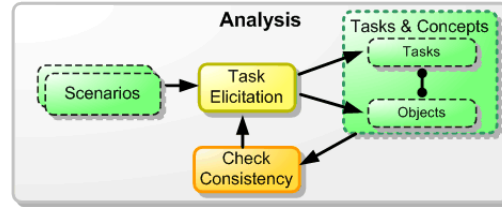


Figure 4.3: The task elicitation step of the analysis phase resulting in a set of tasks, which are linked to object names.

4.2.1.1 Process: Examining scenarios

The analysis phase starts by examining the scenario descriptions for all actors relevant for reaching the overall goal. After all relevant actors have been identified, we collect all activities (usually verbs) that are relevant for reaching the defining goal. We call all activities that are relevant for reaching the overall goal tasks. During the task analysis, four different types of tasks can be identified: User tasks describe cognitive processes of the user and application tasks specify internal processing actions of the computer. Interaction tasks describe activities that are required to be done by both, the user and the system through interacting with each other and finally abstract tasks describe activities that are not completely specified so far. Tasks are structured by identifying sub-goals combining a set of logical related tasks together. After all relevant tasks have been identified and structured to form a hierarchy of tasks, we continue by selecting the objects relevant for each task’s performance. One object can be associated to several tasks and one task can manipulate multiple objects.

Like depicted by figure 4.3 a set of scenarios is used during the task elicitation step. Each scenario describes only one possible sequence through the set of identified tasks and objects. During the analysis phase all tasks and the associated objects are described by a unique and self-describing name and a small description. An example of such an scenario is presented in the case study of chapter 7.

4.2.1.2 Notation

Beneath a pure textual and therefore informal analysis based on scenarios, other approaches exist that can be utilized during the analysis phase. An approach that addresses similar problems is the Use Case Model mainly used during the requirements analysis phase by Jacobson et. al. [Jacobson, 1992]. One advantage of use-case-based analysis is the explicit distinction of what exists outside the system and what should

be handled by the system. The former is identified using actors whereas the latter one describes a use-case, which is defined as a sequence of interactions in form of a dialog between the actor(s) and the system. UML offers a notation that allows a developer to express some aspects of a use case in a more formal way. Both, the scenario and the use-case-based analysis share the problem in the lack of formalities of their definitions. Cockburn [Cockburn, 1997] for instance has identified 18 different definitions for use-cases but agrees with the definition by Jacobson [Jacobson, 1992] (section 6.4.1.):

“A use case is a complete description of a system (or subsystem) function from its start to the end to offer some results of value to an actor. It is expressed as a sequence of messages one or more actors send to the system and its responses. A use case includes the normal mainline behavior as well as possible variants of the normal sequence, such as alternate sequences, exceptional behavior, and error handling. A specific execution case of a use case is called a scenario; each use case contains many scenarios in general. An actor is a role that is played by a user or neighboring system that exists outside of the target system and communicates with it. When a user plays more than one role, we regard that there are as many actors as the roles. A set of all use cases regarding a system specifies its complete functionality.”

In recent years the UML specification of a use-case received a lot of critique, for instance [Isoda, 2003] discusses the use-case class and its instance definition that does not conform to UMLs definition of a class as well as the execution control of a use-case instance in UML 1.3. Paterno [Paterno, 1999] mentions the difficulty of identifying use-cases for very large and complex projects and the problems in finding the correct granularity of use-cases for a given application. During our case-studies it figured out that task modeling enables a compact and comprehensible view at different abstraction levels even when analyzing large and complex interactive systems. This is an advantage compared to use-case modeling as the typical concretization happens by using a new use-case that details a more abstract use-case.

4.2.1.3 Testing step

The result of the analysis phase, the task and concepts model can be tested for consistency. This is done by checking the following aspects of the model:

Object propagation Each object of the task and concepts model has to be identified by using a unique name. Objects with the same name that are reoccurringly used by different tasks identify the same object. To ensure a basic object flow, objects that are used by two child tasks and share the same parent task, have to be linked to this parent task as well. Although this can be done by an automated process, the resulting basic task and concepts tree gives a good overview about the kind of information that is required at each level of abstraction. Similar to deciding between which information has to be considered as global or local information for reaching the overall goal, the amount of objects that need to be propagated to

the highest tree levels to ensure an object flow, give a good impression about the actual separation of concerns of the interactive system. Thus, if a lot of objects are annotated near the root node that describes the overall goal of the interactive system, the developer should think about the object assembly and might decide about subdividing the objects into several parts to prevent an object flow over the whole system. The more the object flow within the interactive system can be reduced to parts of the task hierarchy, the more the interactive system can be considered as “*loosely coupled*” regarding the utilized information structures.

Object utilization Each task identified during the analysis phase has to link to at least one object that the task produces, manipulates or requires to perform. If no object is attached to a task, the task is not able to produce something that is considered relevant for the interactive system. In the analysis phase, this should not happen. The only exceptions are tasks that have been marked as abstract and therefore will be refined later on.

4.2.1.4 Tool support

One of the first tools for task eliciting has been proposed by Tam et. al. [Tam et al., 1998]. They implemented and evaluated a tool called User-Task Elicitation Tool (U-Tel) that has been designed based on experiences gained by Wizard-of-Oz experiments [Hix et al., 1993]. Using U-Tel a domain expert recounts several usage examples and transcribes them into a textual description. After that the expert identifies the task terminology consisting of actions, objects and actors. An outline editor is used to segment the original scenario text to form a structure of tasks and sub-tasks. A basic task-grouping functionality can be used to indicate “is-a” and “part-whole” relationships. Finally, sequencing information can be specified by indicating for each sub-task if it has to be performed in a sequence, may be done in parallel or can be repeatedly used. Most of U-Tel’s functionality relies on structuring outlined text, which emphasized out as a fundamental limitation of U-Tel [Tam et al., 1998].

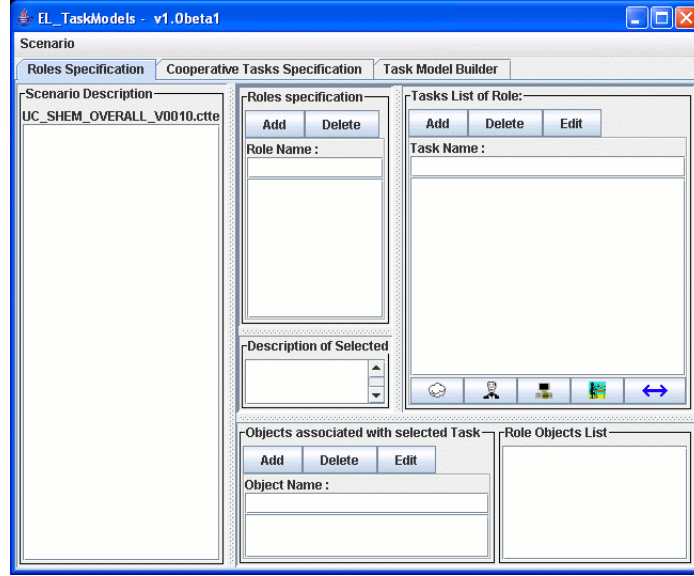


Figure 4.4: A tool for tasks and concepts identification from [Paternò and Mancini, 1999].

Another tool, which addresses the limitation of U-Tel has been proposed by [Paternò and Mancini, 1999]. They describe the EL-TaskModels tool, which is depicted in figure 4.4. Similar to the U-Tel tool the EL-TaskModels tool loads textual descriptions (scenarios) that are examined for verbs indentifying activities, words identifying objects and subjects identifying actors. Each word that is marked as relevant in the textual scenario description can be added to one of the lists, collecting the activities as tasks, the objects associated with each task, and the actors that get abstracted and described as roles. Compared to U-Tel the EL-TaskModels tool enhances the options to further categorize the tasks enabling to specify who has to perform each task: the user, solely the system or an interaction between both, the system and the user. Unlike the U-Tel tool it enables a user interface to graphically structure all identified tasks to form a hierarchy.

Both tools are suitable to support the analysis phase of our methodology as they both end up with a set of structured activities (tasks) that are linked to roles performing these tasks and objects that are required for task performance. In our case studies we utilized the EL-TaskModels tool as it is freely available and has the benefit of more advanced graphical editing functionalities compared to the U-Tel tool.

4.2.2 Design Phase: Task Modeling

The design phase takes advantage of the analysis results and consists of six sequential sub-phases: task modeling, domain modeling, AUI modeling, CUI modeling, Layout modeling and Service modeling.

Since a user-centered design approach requires detailing tasks before objects, we will continue this section by describing the task modeling step first. Although there is an increased evidence that task models are effective in driving user-centered design

[Tam et al., 1998, Wilson et al., 1993], not all developers are feeling comfortable with the idea to think in tasks instead of objects (mostly this involve developers with a strong background in object-oriented modeling). Thus the method does not explicitly require a task-first approach (both the object-perspective and the task-perspective are used within the design phase), but it recommends to start with the task-perspective, in order to implement a user-centered design approach.

4.2.2.1 Process

The Task modeling sub-phase that is illustrated by figure 4.5 requires the result of the preceding analysis phase, the task and concepts structure. During task modeling, tasks get further detailed into sub-tasks and end up in a comprehensive task tree structure. During task modeling three basic activities are done to refine the initial basic task tree: Further introduction of sub-tasks to detail the task tree until basic tasks have been indentified, specification of temporal relationships between all tasks on the same level of abstraction, and further refinement of the object flow by detailing the required input objects and corresponding output objects for each task.

Hierarchical task decomposition is a very intuitive process and very similar to what people are doing in order to solve complex problems by decomposing them into smaller problems. But unlike problem decomposition that usually stops as soon as a problem is small enough to be examined and understandable by a single person, task decomposition targeted to design an interactive application has not a defined final state as it merely depends on the interaction capabilities of the supposed end-devices to be utilized for interaction with the user. Thus, tasks have to be decomposed until they are primitive enough to be even utilized on the most constrained platform. If for instance voice-based platforms are considered as a target platforms, the tasks have to be decomposed until they can be used by a strictly sequential interaction. In the case that only graphical platforms are considered as target platform for presenting the interactive application, task decomposition might end up describing complete forms or even dialogs. Each new introduced task has to be categorized either as an application task (that is performed solely by the system or as an interactive task that includes interaction with the user. Tasks are set to abstract tasks if they are composed of sub-tasks belonging to different categories or if they aren't basic tasks, but require further specifications to be detailed.

Temporal relationships between tasks In addition to the task refinement, temporal relationships between the tasks have to be identified to form sequences of tasks that have to be done in order to reach the overall goal. Temporal task relationships are identified by following a top down approach and are inherited by the sub-tasks. This approach enables a modeling starting from very abstract task definitions to more concrete ones and therefore is well suited even for modeling large interactive systems. Abstract-to-concrete modeling ensures that the designers will not loose their general view of the whole system. Temporal relationships have to be specified between two tasks of the same abstraction level that share the same parent task. Tasks on the same abstraction level have to be associated with at least one other task by a temporal relationship. The

range of available temporal operators and the permitted task connections depend on the utilized notation and the supporting technologies. Its up to the designer to decide about the level of detail the task model is specified. Table 4.1 lists the temporal operators of the ConcurTaskTree notation [Paternò, 1999].

Temporal relationship types	CTT/LOTUS
unordered-lists/variable sequences	Concurrency with information exchange
Choices	Choice
Concurrent operations	Independent Concurrency
Fixed sequence	Enabling/ Deactivation
Constraint linear sequence	Suspend-resume
Cycles	Iteration/Recursion
Optional completion	Optional

Table 4.1: Overview about temporal relations of the ConcurTaskTree notation.

Object flow refinement Together with the task decomposition all objects that are associated with the decomposed tasks have to be decomposed or refined in order to reflect the level of detail that is required for each sub-task. Two different activities of refinement can be done:

1. *Refinement* of objects happens if a sub-task refers to an object of its parent task but requires more detailed information that has not been captured reflecting the abstraction level of the parent task. Refinement is often done by introducing an inheritance relationship for the object that needs to be detailed.
2. *Splitting* of objects is done if a sub-task requires only some part of the original object of the parent task. Object splitting is done for the related object associated with the parent task. A precisely applied object splitting can reduce the amount of objects and the object size especially for the higher abstraction levels and can therefore reduce the complexity of the object flow of the resulting interactive system significantly (especially when designing large interactive systems consisting of a lot of tasks and objects).

During decomposition the object propagation has to be refined as well. Therefore we utilize the concept of *port semantics* like suggested in [Klug and Kangasharju, 2005]. They distinguish between input ports and output ports that are specified in form of an interface for each task and are inherited through the task hierarchy. Input ports define the objects a task has to consume in order to perform successfully. The specification of input and output ports has to conform to the temporal relationships between the tasks to ensure a working object flow.

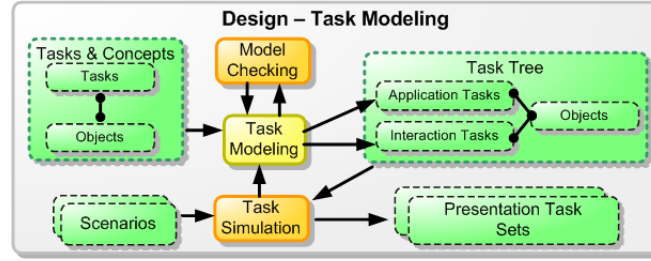


Figure 4.5: The task modeling step of the design phase constructs a task tree that can be simulated to check the compliance with the initial scenarios described during the analysis phase.

Like illustrated in figure 4.5 the result of the task modeling sub-phase, the designed task tree, can be simulated to check for compliance to the original scenario descriptions that have been used as initial input to the task modeling.

4.2.2.2 Notation

The ConcurTaskTree notation has gained a lot of attention in recent years, because it offers a very compact notation and includes well defined temporal LOTUS operators and is already supported by a comprehensive tool that is public available. We have described and evaluated this notation in paragraph 3.1.3.3. Other task-based notations can be utilized for task modeling in order to reflect specific requirements like for instance considering cognitive processes in more detail (which the CTT notation is not designed for). The GTA notation focuses more on modeling collaborative tasks relevant for designing groupware systems. GOMS introduces notation elements to detail cognitive processes. In section 6.2.1 we will propose our own task notation, which is based on the concepts of the CTT notation but is designed to be directly interpreted instead of focusing on tasks analysis and interactive system evaluation support the CTT notation was originally targeted to.

4.2.2.3 Testing step

Testing during the task modeling phase can be done by following two different approaches: First, by doing model checking of the task hierarchy and second by doing a task simulation. The former can be applied to test the consistency and correctness of the task hierarchy, whereas the latter compares the original scenarios with the scenarios that can be simulated by the modeled task tree.

Model checking of the task hierarchy helps to deal with the complexity of the temporal relationships that constraint the object flow between the tasks. Thus, a model is only complete and correct if all tasks that have an associated input port are guaranteed to receive the desired objects through a corresponding output port of a related task [Klug and Kangasharju, 2005]. Depending on the size of the task tree, ensuring the

correctness can result in extensive calculations to prove that every possible activation path between two composite tasks includes at least one task with an appropriate output port.

During the task modeling testing step the user can be involved since testing of a task model is about checking all of the scenarios that served as an initial input to the task analysis against the task tree. Task simulation helps to ensure consistency of the modeled task tree compared to the original textual scenario descriptions. In the case that the interactive system includes a lot of scenarios that have been written by different users or domain experts, inconsistencies between the textual scenarios and impreciseness of the informal format are discovered at the latest during task modeling. The simulation of the modeling state together with the authors of the textual scenarios helps to solve these inconsistencies.

4.2.2.4 Tool support

Since task analysis has a long history in analysis, design and evaluation of operations, processes and interactive applications, a lot of tools are available to support task modeling. The ConcurTaskTree Editor (CTTE) [Paternò et al., 1997] supports the modeling of task trees to form a hierarchy of tasks, where tasks on the same tree level are related by using temporal LOTUS operators. LOTUS is a specification language developed by the International Organization for Standardization (ISO) and was originally developed for the specification of open system interconnection (OSI) services and protocols.

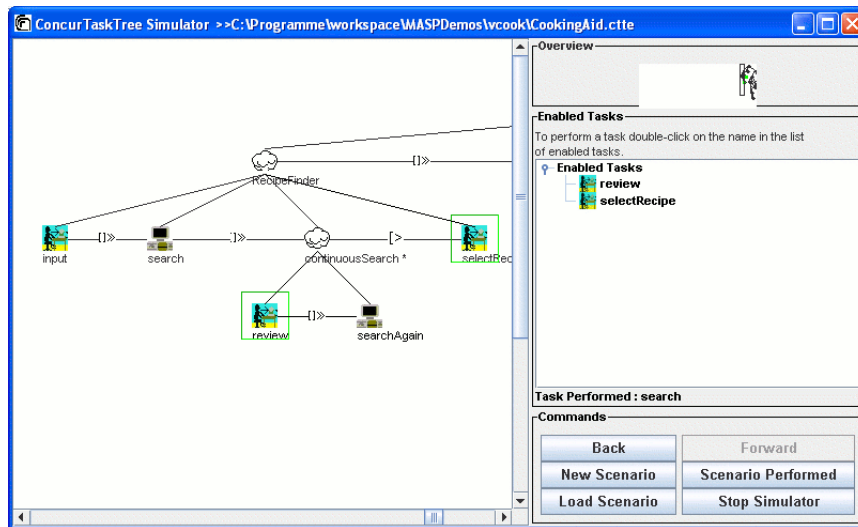


Figure 4.6: The CTTE editor supports interpreting a ConcurTaskTree by offering a simulation tool that allows to check if the temporal relationships between the tasks are correctly set to reflect the original scenarios.

Figure 4.6 depicts a screenshot showing a sub-tree of an interactive application's ConcurTaskTree that is simulated to check if it reflects the initial informal scenarios correctly.

Since a task tree usually unites several scenarios, the simulation has to be done repeatedly for each scenario. Successful simulation can be saved, whereas discovered errors can be directly changed by using the integrated modeling environment of the CTT editor. The CTT editor does not offer an integrated view during task simulation that include the presentation of the initial textual scenarios and the actual task tree, which might ease the comparison of the modeled task tree with the informal scenarios. Further on the CTT editor offers only limited model checking functionalities, which includes a reachability analysis that can be applied to test if a process exists that allows the user to reach a specific task A from a given initial task B. As far as we figured out the CTT editor supports to model an object flow by enabling the designer to specify output and corresponding input actions for each task. But a model checking regarding this object flow, as proposed by [Klug and Kangasharju, 2005] is not supported.

To overcome these disadvantages we implemented both, a Task Tree Editor that supports task analysis and design of interactive systems as well as a model checking of the object flow and a task simulator that is able to generate interactive screen masks by executing our task model notation.

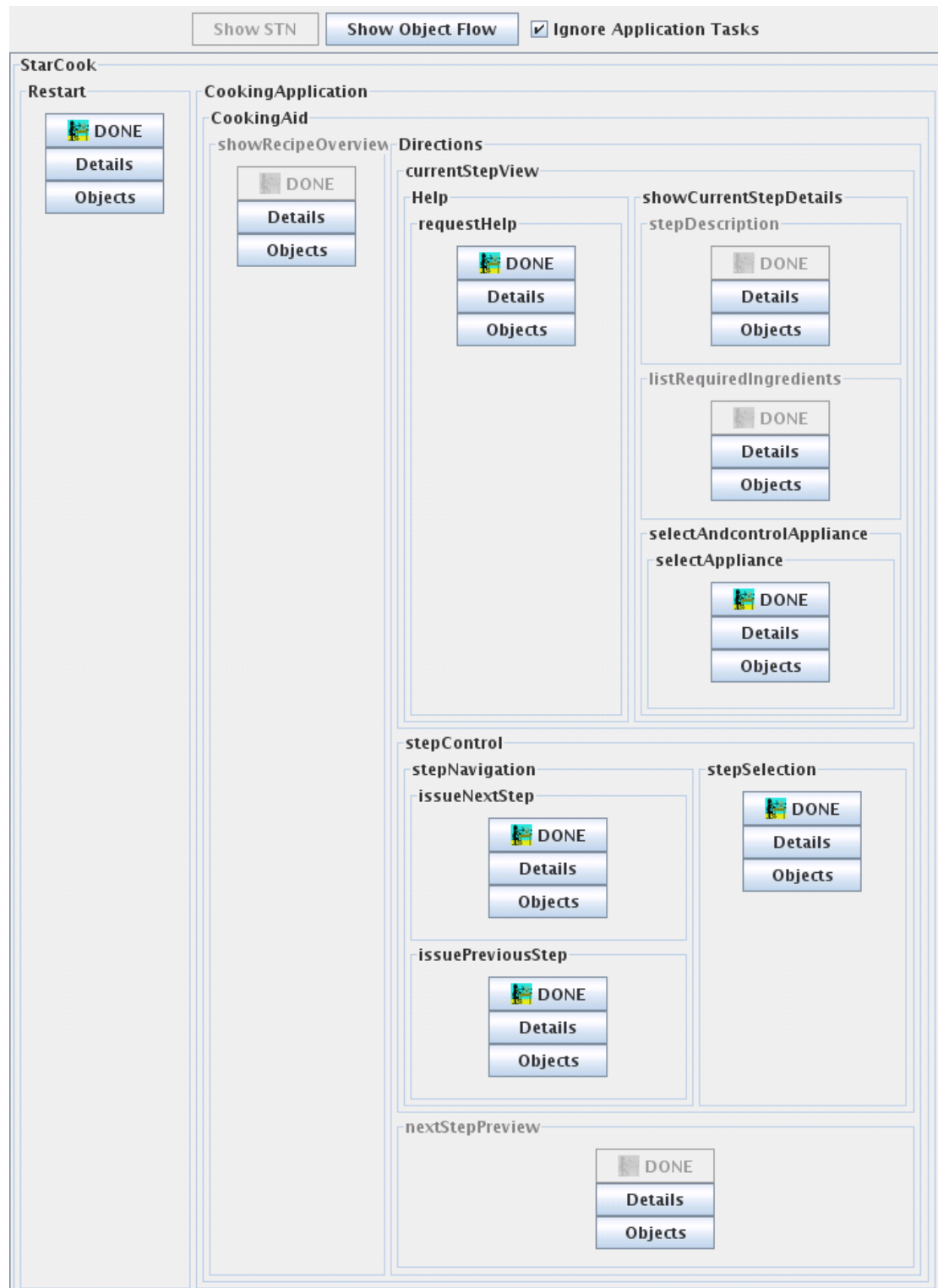


Figure 4.7: The screen for the object refinement in the MASP Task Tree Editor.

Figure 4.7 shows a screenshot of the task simulator tool that has been taken during testing the task model of the cooking application that will be presented in detail in chapter 7. For each state of the application the simulator displays all enabled tasks and lays them out conforming to the parental relationships of the task tree. For a better overview and space usage for each containment level, the simulator switches the orientation of the containers from vertical to horizontal and vice versa.

Each atomic task is depicted as a box and supports three operations: By pressing the “done” button a task can be marked as done, which signals the simulator to layout the next enabled task set. All corresponding objects that a task is able to manipulate can be inspected by the “objects” button” and finally, by pressing the “details” button, more detailed task information containing the textual description, the task type and attributes are visualized.

Beneath the possibility to check the consistency of the task flow against the original scenarios, the task simulator helps to identify deadlocks of the task flow and missing tasks. Deadlocks of the task flow can occur if there is no input interaction task left in a certain enabled task set, which results in an output interaction-only presentation that the user is unable to interact with. Such “output only” situations should only happen if the overall goal of the application has been successfully reached or an error has occurred that can only be solved by restarting the complete application. Like depicted in figure 4.7, *listRequiredIngredients*, *showRecipeOverview* and *nextStepPreview* are realized as *interaction out* tasks that only offer information to the user. For these tasks the done button is disabled and therefore no interaction with the user is possible. The only option to finish these tasks is to succeed by an *interaction in* task that disables the *interaction out* tasks. In figure 4.7 the *restart* task disables all other tasks (which has been defined by using the disabling operator in the task model - like depicted by figure 8.6 in the annex).

By model checking we ensure a consistent object-to-task definition in the task model. The model checking helps on the one hand to discover objects (and the concepts they implement) that are re-used over certain tasks. On the other hand the object operations (declare, create, read, and modify) can be automatically checked for consistency.

All objects that are annotated to the tasks of a task tree must include an object operation. The object operation for each object can be checked to follow the basic sequence (1) declare, (2) create, (3) read or modify by an automated model checking. Additionally these operations are verified concerning the type of a task. Whereas *interaction in* tasks can create, read and modify domain objects, *interaction out* tasks are only permitted to read domain objects. Further on, the declare operation limits the scope of a domain object to a certain sub-tree. When modeling complex applications, concepts that are spread over the whole application can be easily identified since they are declared close to the most abstract root task (like for instance the *selectedRecipe* object in figure 8.6).

In general, objects that are spread over the whole application should be broken down or disassembled to reduce the information flow along the whole application and the task sequence should be re-thought in cases where one concept is used several times but not subsequently. Whereas the former leads to interactive systems requiring a lot of initial domain knowledge to be utilized the latter ends requiring the user to switch between

different concepts.

By proceeding the task modeling sub-phase and entering the domain modeling sub-phase, a change of the development perspective is required to combine both, 'traditional software-engineering' approaches concentrating on identifying the objects first and a user-centred perspective identifying tasks before the objects. The method therefore starts with the user perspective and changes during the domain modeling sub-phase to a system view in order to consider an object-oriented view over the interactive application.

4.2.3 Domain modeling

Different to the task modeling that is used to identify the general processes of an interactive system, the domain modeling concentrates to model the static data structures that are required for a successful task performance.

The domain model is taking into account the application tasks and the domain objects and offers an object-oriented perspective about the interactive application. Since at that state of the development process these objects are only known by their names and a short description telling what they are expected to specify and contain, the next thing to do is to look for already existing concepts that can be re-used. In this case all the services and functionalities that are based on these concepts are potential candidates to be re-used as well. Specialized concepts of the developed application have to be modeled using a modeling editor that supports modeling of static structures such as entity-relationship models or class models.

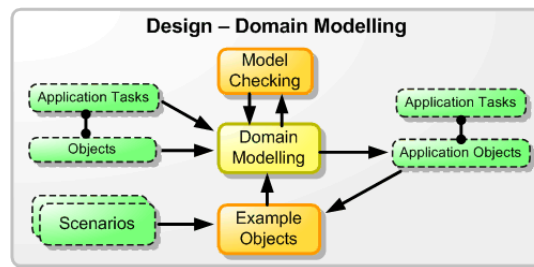


Figure 4.8: The domain modeling step of the design inspects the application tasks and objects for re-use and designs missing concepts.

4.2.3.1 Notation

All new or proprietary domain concepts that cannot be derived by re-using already existing concepts have to be modeled from scratch. This includes selecting a suitable modeling notation and deciding about a technology to implement these concepts later on. For both, notation and technology, a lot of different options exist. Regarding modeling notations UML and entity-relationship diagrams are actually two of the most prominent and widely accepted options to design static data structures. There already exist a lot of alternative

technologies to implement the designed domain concepts. Technological options include various programming languages that offer data structuring types, databases or ontology repositories.

Independently of the notations and technologies utilized, we switch the modeling perspective from a user-centric perspective by identifying the tasks before the objects during the task analysis to a system and service-oriented perspective by modeling the application objects before the interaction objects.

4.2.3.2 Process: Modeling application objects

To generate a system-oriented perspective, we collect all application objects that have been identified during analysis and task modeling. For each application object we look up all associated application tasks where an application object serves as input or output or both. Every application task is checked if it has been already implemented or should be provided by an external party. Existing application tasks and application objects have to be evaluated to be re-used. If no services or concepts can be re-used or re-usability is not important, the designer has to model the concepts for each application object and check their suitability to serve as an input object (parameter) or output object (return value) for an application functionality.

Mapping between task model and domain model Since the domain model has to be constructed based on the information offered by the task model, the mapping between both models is constructed by linking the object names that have been specified in the task model for every application task with the application object names that are modeled in the domain model. Looking from the direction of the task model to the domain model, the mapping has to ensure, that every application object that is considered in the task tree as an input or output object can be read from and written to the domain model. Looking from the domain model to the task model, the domain model should ensure that only application objects are stored within the domain model that are referenced at least by one task of the task model. If a task is active during interpretation of the task model, the domain model has to ensure via the mapping that the active tasks are always aware of changes to the domain model's application objects.

4.2.3.3 Testing step

Testing during the domain modeling phase involves model checking and derivation of example objects to check the completeness of the domain model. To preserve consistency between task and domain model during the design-time, model checking includes verifying that every object that has been specified as relevant for performing the application tasks has been modeled in the domain model.

Depending on the technologies that the domain model is based on, the consistency checking between task and domain model can get very complex. If for instance the domain model supports inheritance relationships the objects can be detailed by utilizing the inheritance relationship through the levels of the task tree, the model checker has to

support introspection of the domain object structure to check the inheritance relationship as well.

Beneath testing domain-model and task-model consistency, the scenario completeness can be checked by deriving example objects that are created by instantiating the application objects. Thus, for every task all the information that has been identified by the associated application objects and that is required for the task performance is entered based on the informal scenarios. As in the domain modeling phase only application objects are taken into account that reflect more or less the data structure from the viewpoint of the systems back-end functionality, tool support is required that is able to handle the user input of complex or decoded data types (like for instance hex values).

4.2.3.4 Tool support

Depending on the chosen technologies and functionality that is offered by external service providers there exists a more or less comprehensive tool support for domain modeling. During our case-studies that we did to evaluate the methodology, we often had to consider more than one technology for domain modeling, since often already existing services that provide their own data models have to be integrated. Thus, often not only one tool is used for modeling the domain model, but a set of tools supporting different notations. For instance if a database should be connected to serve as a data source for the domain model, entity-relationship models can be used to model the structure of the domain model and already existing tools include model-editors and tools to fill the database with content.

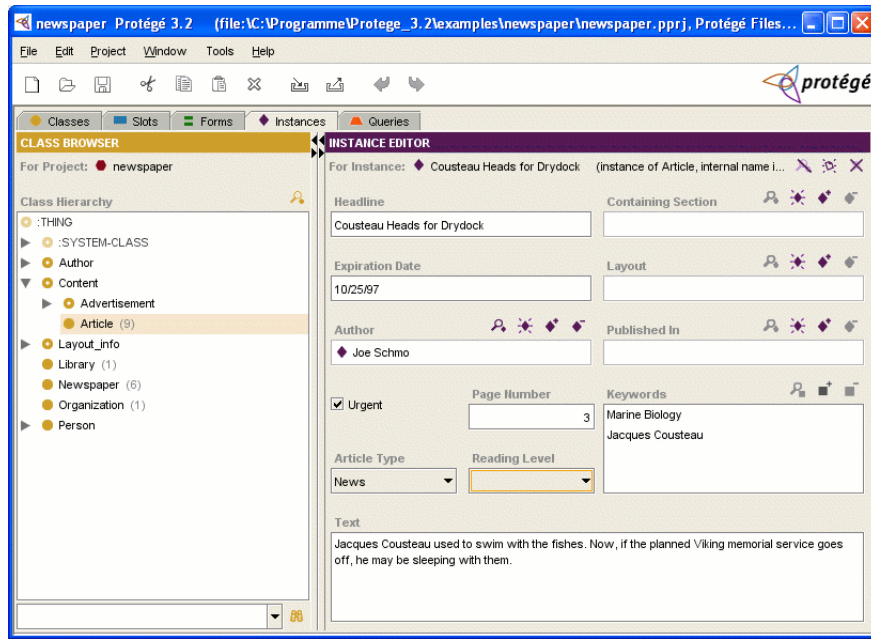


Figure 4.9: The Protégé editor can be used to model various data structures like for instance ontologies or class diagrams. The screenshot depicts an automatically derived form that has been generated by inspecting the semantics of the underlying ontology.

Recently ontology notations like OntoWeb [Hartmann and Sure, 2004] and OWL² receive a lot of attention as they offer mechanisms supporting the semantic web by enabling to specify rich semantics describing data in a way that is can be processed by a computer. Figure 4.9 depicts the user interface of the Protégé editor [Noy et al., 2001] that can not only be used for data modeling, but automatically generates forms that can be used to create the example objects to check the scenario completeness of the domain model. Depending on the expressiveness of the language utilized for modeling the data structure the Protégé editor generates more or less advanced user interfaces. This includes considering constraints and relationships between classes.

Like illustrated by figure 4.1 after the task model and domain model have been designed and tested, the abstract user interface can be derived. This involves a transformation that considers both, the task model as well as the domain model.

4.2.4 Abstract User Interface Modeling

Following the abstract user interface model definition of [Limbourg, 2004], “an Abstract User Interface (AUI) model is a user interface model that represents a canonical expression of the renderings and manipulation of the domain concepts and functions in a way

²OWL Web Ontology Language Semantics and Abstract Syntax, W3C Recommendation; Online at <http://www.w3.org/TR/owl-semantics/>, last checked 12/08/08

that is as independent as possible from modalities and computing platform specificities”. We derive the abstract user interface by considering the information gathered so far from the domain modeling and additionally utilize the task model’s structure and application object flow to derive the abstract user interface.

4.2.4.1 Process

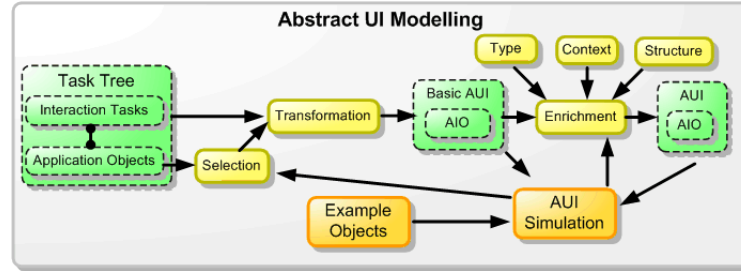


Figure 4.10: The AUI modeling step of the design creates an AUI based on selection and enrichment techniques.

Figure 4.10 illustrates the AUI modeling sub-phase, which requires the interaction tasks of the task model together with all application objects that are marked as input or output to an interaction task and have been modeled by the domain model. Both, the task tree and the associated objects are subject to a transformation that constructs a basic AUI for each interaction task and only considers the application object attributes that have been previously marked as relevant for the user interaction to form an abstract interaction object (AIO). All AIOs that are relevant for an interaction task are grouped together to compose the basic AUI. We then enrich the basic AUI by transforming data types that might have been encoded within the source application object (for instance by hexadecimal values that are often used to represent color values) or that are too fine grained for user perception (for instance values expressing durations in milliseconds). By adding contextual information such as labels and help texts to the basic AUI we can support the user in understanding the AUI of an interaction task. By refining the basic AUI structure and by grouping related AIOs, the run-time system is supported to decide about what AIOs must be presented together and which groups of the AUI can be migrated independently from the remaining parts to other platforms. Since we derive the AIOs from the application objects, it simplifies matters to retain the mapping between both types of objects.

The task model structure and application object flow that have been specified during the task modeling step are used to generate the foundations of the AUI structure and should not be manipulated during AUI modeling to preserve the consistency of the AUI to the task model. In case that the automatically generated basic AUI structure figured out to be not as expected, this usually points out a modeling mistake in the task level model and can be fixed by going back and reconsider the results of the task modeling step. Based on the automatically derived basic AUI structure the designer has to refine

the AUI by applying two different activities to the basic AUI structure: selection and enrichment. The former one is the general mechanism to derive the AIOs from the application objects, whereas the latter concerns the complete AUI structure and all of the AIOs that are referenced by the AUI specification. The next paragraph introduces both activities in greater detail.

Activities used for AUI derivation

1. *Selection activity*: The domain model is designed from a system perspective that is considered to describe a domain on a level of abstraction that is suitable for computational processing. Thus, the domain model structures all data that is relevant for the domain of interest of the application, for the internal processing, and for storing all relevant states. These structures are designed to ensure consistent application states and are optimized for an efficient access like for instance done by data table normalization that subdivides tables and introduces internal index keys to relate subdivided tables. A selection activity first has to separate internal data structures from the data structures relevant for user interaction and second has to segment these data into portions that can be handled by the user interacting with the system.
2. *Enrichment activity*: The structured information of the domain model that has been selected and portioned into portions that can be managed by the user during interaction with the system is typically not self-evident so that the user can handle it without further information. This has several reasons: First, the result of applying a selection activity takes the data out of a context that is important to handle it. Second, most of the actual technologies to implement a domain model miss a way to add a meta-description of concepts. Therefore often a separate documentation is maintained that describes the domain model semantics that cannot be expressed by the domain model itself. Third, not only the data structure but also the actual task is required to understand the context of an AIO. Finally, data types considered in the domain model cannot generally be used for user interaction because they are encoded and need to be interpreted (like for instance color values that are often modeled as hexadecimal values) or have a very fine graded resolution that is beyond the human's capabilities to handle (like for instance deciding about intervals that happen in micro-seconds).

There exist three ways of enrichment:

- a) *Type mapping*: transforms typed data into human understandable types, which is mostly a reduction of complexity like boolean to string or long to integer type conversion and enables type checking in the user interface engine (additionally to the error handling in the application's functional core).
- b) *Context enrichment*: Most of the structured information from the domain model is data only and does not include information how the data is processed and misses explicit semantics. Therefore context enrichment is required to

inform the user about how to interpret the presented data. This involves the distinction between input/output information, as well as adding labels to the data or adding text that explains what needs to be done by the user. Beneath adding information that is relevant for the user, design patterns as well as further information about, which interactors are most useful for presenting the domain data can be added by the context enrichment.

- c) A *structuring concept*: domain model data is often already structured compliant to a concept like tables, classes or ontologies. This structural semantics can be used to generate user interfaces that implement this structure. As ontologies provide a lot of semantics like concepts to express different kinds of relationships or constraints, most of these concepts are helpful for automated user interface generation. But often these structures are defined on a very fine grained level. On a higher level of abstraction the user interface needs concepts for grouping related data (that it receives from initial unrelated ontologies like for instance provided by different backend services). In the concrete user interface these structuring concepts are often used for layouting the interface based on the capabilities of the end device (like for instance to adapt the user interface to the screen size of small-sized devices).

In the following sections we describe one possible approach for AUI modeling consisting of a notation and a tool to realize the selection, transformation and enrichment steps. The AUI model is derived from a domain model that is structured by using an ontology language to specify its application objects and a task model that is modeled using the CTT notation. Different to other approaches that derive the AUI from a task model, the derivation from the domain model enables preserving already existing semantics of the domain model to be considered for generating the user interface. In this way the presentation of domain concepts can be done once and reused consistently.

Abstract Interaction Object derivation Interaction objects are derived from application objects since all interaction between the user and the system usually ends up with the systems computation (via application tasks) of the user's inputs and often includes to communicate the computation results back to the user (via interaction tasks). Thus, the application objects are required as input and output for performing the application tasks and further on form the foundation for modeling the interaction objects, which describe what part of the data is relevant for the user to be presented and manipulated via a user interface.

As a result from the analysis phase only the object's names and short textual descriptions are available for each interaction task. To model the interaction objects based on these descriptions we identify for each interaction task all relevant input application objects as they should cover every possible data that might be relevant to the user. Next, we collect all application objects that are marked as output objects as they specify the data that the system is required to capture in order to perform its computations. Further data structures that cannot end up in an application object should not be modeled, since this data is not required to succeed the user's goals. If however interaction objects are

modeled that contain data that cannot be completely mapped to corresponding application objects, this has to be considered as an indication that some application objects and corresponding application tasks should be re-designed to include these information.

Different to [Paternò and Mancini, 1999] we derive interaction objects during the AUI modeling, since this approach eases generating and handling the connection between both types of objects and helps to simplify the mapping between application objects and interaction objects, since the latter ones are based on the former ones.

To derive interaction objects based on the data structure of application objects, the following steps have to be done by the developer: Firstly, by an attribute selection all attributes that are relevant for the user interaction are marked. Secondly, all marked attributes have to be restructured as they might be:

- selected from different application objects and should be merged together into one object or
- selected from a complex application object structure that contains for instance internal composition and inheritance relationships and does not reflect a structure that is suitable for presentation.

Finally, the resulting interaction object structure has to be enriched with additional information helping the user to understand the original application object context by offering help texts, and attribute labels describing the attributes to the user.

4.2.4.2 Notation

The notation for modeling the AUI model by specifying the selection and context-enrichment is separately described in detail in the MASP language section 6.2.2.

4.2.4.3 Testing step

Testing of the AUI can be done in parallel to the AUI modeling process by simulating the AUI of the interactive system. As illustrated in figure 4.10, the simulation can be done directly after the basic AUI has been modeled or by simulating the final AUI model. At this state it is possible to present the targeted audience of the interactive application a first impression how the described processes will be supported by the interactive system. As far as we know, no tool is available to simulate the behavior of the user interface based on the abstract user interface. IdealXML [Montero and López-Jaquero, 2006] is a tool that supports user interface derivation based on a task model and a domain model, but does not focus on AUI simulation. Klug [Klug and Kangasharju, 2005] interprets the task tree similar to our approach to derive an interactive application but does not address a domain model.

4.2.4.4 Tool support

To support the AUI modeling step, we developed the MASP Builder tool, which enables an interactive selection, enrichment, and structuring of the AUI model based on the

domain model. Further on, during the AUI modeling process, the actual state of modeling can be tested by rendering and previewing the user interface.

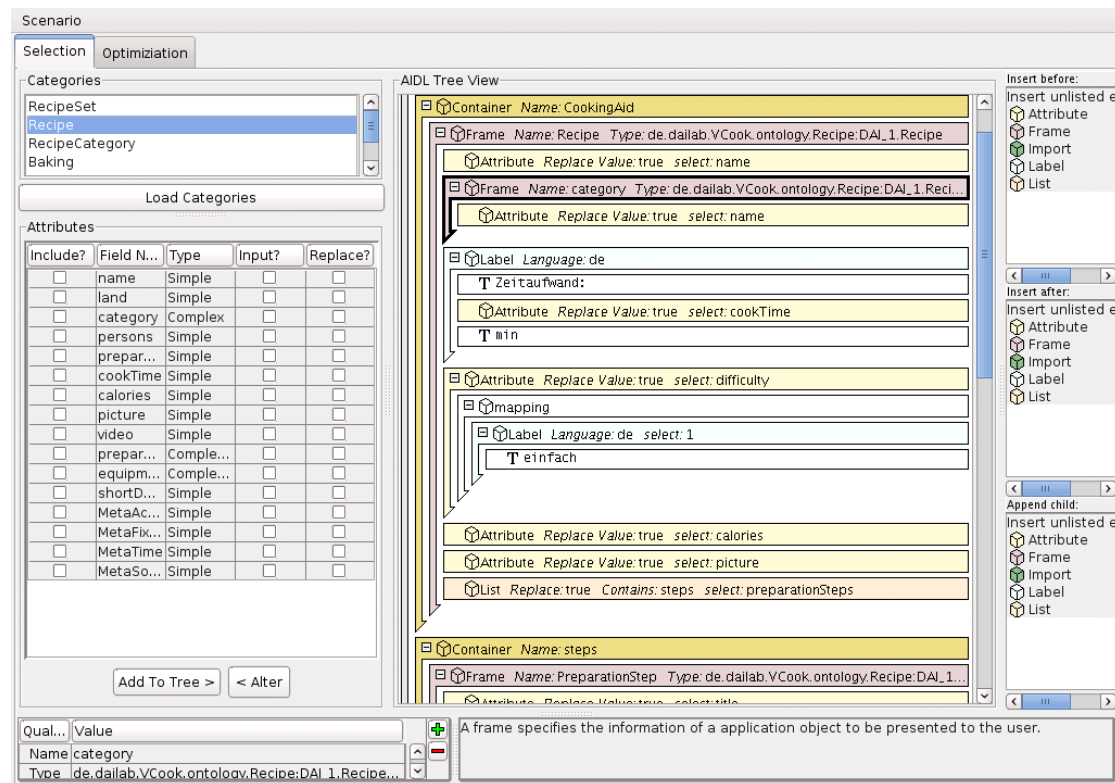


Figure 4.11: The MASP-Builder tool supporting AUI modeling.

Figure 4.11 shows a screenshot of the MASP Builder tool. By the two lists on the left side the concepts of the domain model can be selected and inserted to the main view in the center, that displays the AUI model in form of a tree view. This tree can be enriched by dragging and dropping AUI elements into the tree that are listed in the boxes to the right. At the bottom a context sensitive help is presented to support the designer.

By following the AUI modeling process depicted in figure 4.10, we start by using the MASP Builder to collect all objects that the backend services produce and that should be considered for the presentation of the user interface. This also includes loading their associated data structures (in our case we use ontologies) in the MASP Builder tool. The data structures are listed by the “Categories” list view in the upper left corner of the MASP Builder tool in figure 4.11.

The developer can browse through the attributes of all data structures and can select the attributes that should be presented to the user (the selection activity). Therefore the developer marks relevant attributes in the “include” column of the list view in the middle-left of the MASP Builder. For each selected attribute a type mapping can be set that includes marking the attribute as manipulable by the user (the “input?” column) or the

definition of value to string mappings (for instance linking values to a more descriptive text). For attributes that should only be entered by the user but that should not display their actual stored data to the user, the “replace” column field of the attribute can be unchecked. After the developer has selected the relevant attributes the AIO statements can be generated and added to the AUI tree by pressing the “Add To Tree” button. The resulting tree of the selected attributes is depicted in the main area of the tool.

In case that the developer has selected a complex attribute (for instance containing a composition), the corresponding AUI selection statement that the tool generates in the tree view can be selected to be further detailed. Therefore, by pressing the “Alter” button the MASP builder looks up the structure that the composed attribute consists of and presents all these attributes to the developer to detail the selection for the complex attribute.

Different to the selection activity that is based on the domain model structure, the enrichment activity concentrates on adding further information to the AUI that is not available in the domain model (except for the type mapping). The MASP Builder tool supports the enrichment activity by listing relevant AUI enrichment statements by three list view boxes that are depicted in the screenshot of figure 4.11 at the right. To support the developer, only those enrichment statements are listed that are relevant in a certain situation. Enrichment statements can add textual information, introduce labels for attributes and can be used to restructure the data representation in the user interface. For each statement that the developer selects, a context-sensitive help is given at the bottom of the MASP Builder tool. New enrichment statements can be added to the AUI by dragging and dropping them into the AUI tree. By implementing the restructuring activity and introducing further containers to specify the relations between the different domain model structures, a data presentation structure can be defined that can be efficiently adapted to differently sized end devices and interaction sequences later on. The AUI model statements will be described in greater detail in the corresponding language model in section 6.2.2.

4.2.5 Layout modeling

Interactive applications must be able to support different context-of-use scenarios. This includes adapting their user interface seamlessly to various interaction devices or distributing the user interface to a set of devices that the user feels comfortable with in a specific situation. Such adaptations require flexible and robust (re-) layouting mechanisms of the user interface and need to consider the underlying tasks and concepts of the application to generate a consistent layout presentation for all states and distributions of the user interface.

The broad range of possible user interface distributions and the diversity of available interaction devices make a complete specification of each potential context-of use scenario during the application design impossible. Specifying the interdependencies between the user interface components using constraints is a common approach to address these issues and nowadays constraint solvers can calculate hundreds of constraints in a reasonable amount of time. To our knowledge there is still an approach missing that

supports designers of a user interface in generating these constraints based on the design specifications. A manual constraint setup has two disadvantages: first the pure amount of constraints that is required even to address small interactive systems is hard to handle and second the fault tolerance of the constraint setup is complex to attain. Even one single constraint that is not properly specified can destroy the complete layout in a specific situation that has not been considered by the designer during the development process.

The layout of graphical user interfaces comprises two aspects. On the one hand it structures components of the user interfaces and on the other hand it follows aesthetic aspects. We will focus on the former aspect, whereas the latter can be addressed by manual manipulations of the pre-generated layout later on.

We introduce a concept for a model-based user interface layouting that differs from previous approaches in two general aspects:

1. We interpret the information from already existing user interface design models, such as the task tree, the AUI, the concrete user interface model, the domain model and the context model for deriving the user interface layout. Therefore we propose an interactive, tool-supported process that reduces the amount of information that needs to be specified for the layout. The tool enables designers to comfortably generate these interpretations by defining statements and subsequently apply them consistently to the design models for all screens of the user interface.
2. We shift the decision about which of the statements are applied to generate the final user interface layout from design-time to run-time to enable flexible context-of-use adaptations of the user interface layout. This allows us to describe new context-of-use adaptations of the layout without the need to change the application itself just by describing the layout characteristics of a new platform or a new user profile.

4.2.5.1 Process: The Layout Model Derivation

To derive a layout model the designer has to specify interpretations of the other design models, such as the task tree, the AUI, the concrete user interface model, the domain model and the context model. In general two different interpretations are possible: First, layout interpretations that are explicitly specified for one user interface and second, layout interpretations that are defined independent of the user interface by interpreting pre-defined context information like for instance addressing layout adaptations for specific devices and users or that are only valid in a specific environment. For each new interpretation that is written into the layout model in form of a layout statement the designer can initiate a simulation to preview the result. The simulation positions the individual user interface elements based on the specified layout model statements for all screens and context-of-use scenarios that are known at design-time.

Based on the task, abstract and concrete user interface models a layouting model can be designed. Since the design models offer a lot of semantics about what the application is about and what needs to be presented to the user for interacting with the system, a layout model can be created based on interpretations of the design models. Whereas a model-based layout derivation cannot substitute a manual design (which we consider

as an art), it has two main advantages: First well defined model interpretations can generate a consistent layout, if they are generally applied for the complete application and second, for context-of-use situations that have not been explicitly addressed by the application design, such as new interaction device or user interface distributions.

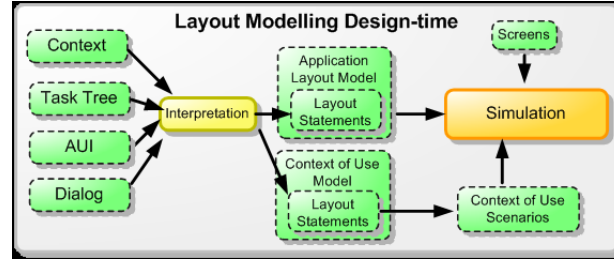


Figure 4.12: In the layout modeling step layout statements are specified and selected that interpret the tasks, AUI, dialog and context models to derive a layout model for graphical concrete user interfaces.

Following our method the layout modeling has to happen after the concrete user interface models have been designed. Since layout modeling is done for the graphic user interfaces only, merely the concrete user interface models describing graphic user interfaces are relevant for the layout model generation. Figure 4.12 illustrates the activities that have to be done to derive a layout model.

The layout modeling ends up in a domain independent layout model that specifies the containment, the size, the orientation and the order relationships of all individual user interface elements. Therefore we do not want to specify the layout manually for each targeted platform and do not rely on a set of standard elements (like a set of widgets for instance) that have been predefined for each platform.

Layouting Statements The main activity that has to be done for deriving the layout model is the specification and selection of suitable design model interpretations.

Like depicted by figure 4.13, we classify the properties of the layouting statements by six axes: (1) the characteristic of the resulting layout primary addressed (containment, orientation, and size), (2) the design models used for the constraint generation, (3) the context-of-use information, (4) the addressed scope, (5) the type of condition and finally (6) the priority value.

The following paragraphs discuss these axes in greater detail.

Layout characteristics We identified four of these characteristics that can be used to specify the layout of a graphical user interface: The containment, the order, the orientation and the size of the user interface elements.

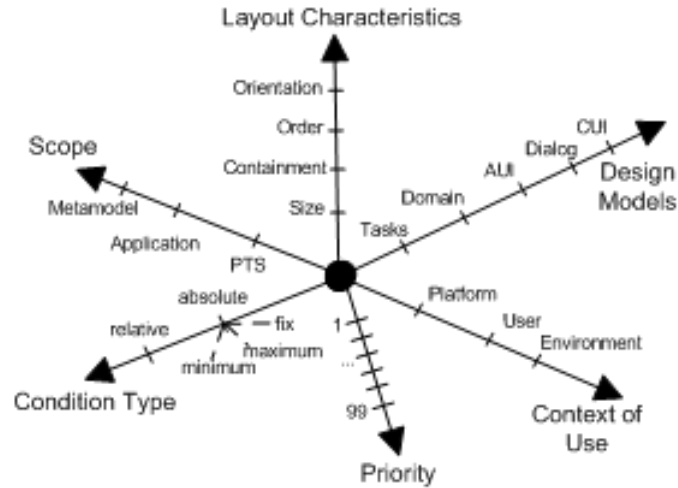


Figure 4.13: The six axes of the space of properties of the layout statements.

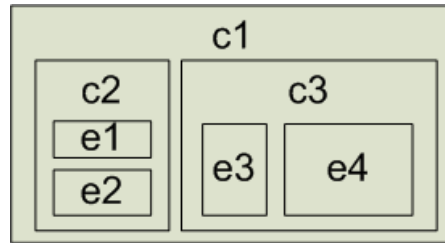


Figure 4.14: Exemplary sketch of a user interface layout.

Like illustrated by figure 4.14, the containment describes the relation between two basic types of entities: Containers (like **c1**) consist of a set of nested containers (**c2**+**c3**) and nested elements (**c2** contains **e1** and **e2**). Elements can present information to the user or enable the user to enter data to the application and cannot be decomposed any further. Additionally a layout describes an order of elements (for instance from left to right and from top to bottom: **e1** before **e2** and **c2** before **c3**). The orientation distinguishes between elements that are oriented horizontal or vertical to each other (for instance **e1** vertical to **e2**). Finally the size specifies the width and height of containers and elements (for instance the width of **e3** is $\frac{1}{2}$ of the width of **e4**).

Like illustrated by figure 4.13, these characteristics can be described at different levels of abstraction that are specified by the design model axis. Thus, containers can represent tasks that are decomposed (by nesting of containers) until atomic tasks have been identified, which are modeled as elements. By the same way on the AUI level, an interaction object that specifies a list of selectable elements can be described by modeling the list as a container consisting of a set of elements.

Design Models Interpretation Design models are used to specify the interactive system on different levels of abstraction. We interpret the information of these models to derive the user interface layout. A task model, a domain model, an AUI, and the concrete user interface model are typically part of a model-based user interface design. Beneath the task model's hierarchical structure that can be used to derive a basic containment structure for the layout [Gajos and Weld, 2004] several other information can be derived: For example, the sum of all atomic tasks related to the task tree depth or related to its width can be used to balance the presentation size of the tasks. The CTT notation [Paternò, 1999] categorizes interaction tasks into “edit”, “control” and “selection” tasks. These task information can be addressed differently related to the context-of-use, for instance by prioritizing those tasks that require user input. Looking at the abstract user interface model, the interaction object type can be used by a layouting statement to derive an orientation. In this way for instance navigational elements can be set vertically or horizontally depending on the menu level, whereas selection elements can be oriented vertical for a large amount of elements and horizontal for small amounts by a layouting statement.

Condition Type Each statement describes either an absolute condition (minimum, maximum, or fixed) or a relative condition that relates two or more elements. A relative condition targeted to the orientation characteristic is for instance: “e1 over e2”, regarding the size a relative condition can specify for instance “e4 double the width of e3” and finally regarding the containment it has the form of “c3 contains e4”. A maximum statement specifying an absolute condition can be used to specify a column layout where elements are wrapped to the next row after a specified amount of elements is exceeded. Further on, a maximum statement can restrict the maximum size of an element (regarding its height or width) or can limit the maximum number of elements that a container can consist of, which results in generation of new containers if the limit gets exceeded.

Application Scope Each statement has a fixed scope to address every application (application independent statements), the whole application, a set of reoccurring elements or a specific screen to handle very fine grained design requirements. Application independent statements are used to characterize context-of-use adaptations that are required to be considered when layouting for a specific device (such as specifying the screen size limitation, or the minimum size of control buttons for a touch screen). Application wide statements help the designer to generalize design decisions and maintain consistency as layouting decisions can be modeled just once and are automatically applied for each re-occurring situation. The more global than local statements have been defined the better is the robustness for contexts-of-use changes and the better layout consistency can be expected. Finally a statement can be limited to address a single screen to fine tune the layout for aesthetical reasons or to refine an application wide layout statement.

Context-of-Use Scope Context models describe the user, who has preferences and demands for the actual situation, and a set of devices that the user likes to use. A layout

statement can be specified to be relevant for a specific context-of-use situation only. For instance in an environment that supports location tracking the distance of the user to a device can be used to scale the control tasks of the user interface. In the former case the control tasks are sized small if the user has no way to control because of his distance to the display, whereas in the latter case the control tasks are sized to meet a pen or a finger print respectively.

Strict order by Priority The priority is required on the one hand to support a general-to-specific layouting approach and on the other hand to prevent the generation of conflicting layout constraints. Thus, general layouting principles, such as described in style-guidelines or given by a corporate design can be generally defined and overwritten to address more specific situations later on. We address these aspects by specifying a strict order in that the layout statements are evaluated to generate the constraints that we indicate by the priority property.

We can conclude that deriving a layout model from the other design models of a model-based approach can be a big benefit, because it allows the layout model to design decisions of the other models in a consistent manner. Furthermore a model-based layout reduces the information that has to be specified specifically for the layout model as a lot of information is already available. Finally a layout derivation can enhance the robustness of the user interface to unknown context-of-use changes the more global model derivations for the layout generation can be identified. To realize such a model-based layout generation that is based on model interpretation we require (1) an efficient way for the designer to select suitable model interpretations for generating a layout, (2) a process that eases the identification of global interpretations to enforce the layout's consistency and robustness against context-of-use changes and finally, (3) a model-based run-time system that can evaluate these interpretations in an efficient manner so that layout adaptation of a user interface is possible at run-time. We introduce our model-based layout editor in the next section that we implemented to address requirements (1) and (2), and describe how we realized the layouting in our run-time environment, by introducing the Multi-Access Service Platform (MASP) to adapt to context-changes (3) in the next chapter.

4.2.5.2 Tool and Testing Support: Layout Model Generator

Using the layout model generator the designer has to initially load all existing design models of an interactive application such as the task tree, the domain model and the AUI as well the concrete user interface and the context model that contains device capability descriptions and the preferences of the user.

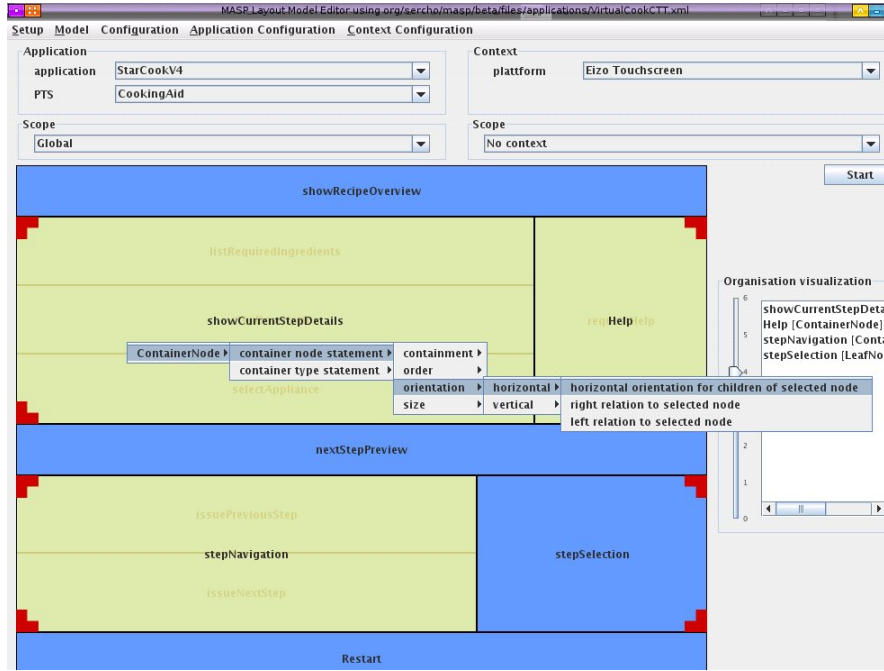


Figure 4.15: The MASP Layout Model Generator.

Figure 4.15 shows a screenshot of the editor: Using the pull down menu button in the upper left corner, the designer selects an application configuration consisting of all design models for that she wants to derive a layout model. As the AUI model contains a set of dialogs, where each dialog consists of a set of elements that should be presented simultaneously, the generator enables the designer to browse through all the screens of the application. Each screen consists of a set of individual user interface elements that should be presented to the user on a single device. User interface distributions can be defined as a set of separate screens where each screen represents a part of the user interface for a specific device.

As the result of the laying out process, the layout model is visualized by a box-based preview that represents each individual user interface element as a box. By the box-based preview the designer gets an impression of the resulting user interface concerning the individual element sizes, containment-, order- and orientation-relationships. The simulator scales the preview in order to comfortably support layout modeling for large displays but considers the aspect ratio of the targeted device.

Whereas the designer defines all of the model interpretations by using a context menu that is related to the box-based simulation area, she can set the application scope (global, application or screen specific) and select the context-of-use for that the statement should be addressed by the two pull down menus above the simulation area.

The interactive and tool-supported process follows a sequence of steps:

1. The designer decides about the layout characterization that the statement should address: the containment structure, the element order, the orientation or the size.

2. The designer interprets
 - a) design model information (such as the AUI type: input, output, control, or selection task or the CUI type)
 - b) context model information that requires a layout adaptation.
3. The designer can visually weight a relational statement. for instance relate the size-ratio between input and output elements in general or specify size relations between two specific boxes.
4. The Model Generator automatically applies the new statement consistent to the design models to all screens of the applications (limited by the scope of the statement).
5. The Model Generator updates the boxed simulation area to reflect the new layout for all screens and all actually supported context-of-use scenarios of the user interface layout.
6. The designer checks the result and manipulates the order of the statements.

In order to ease the identification of global interpretations and to enforce the layout's consistency and robustness against context-of-use changes (requirement 2), we implemented an abstract-to-detail slider, which is depicted to the right in figure 4.15. The slider allows the designer to browse through the nested boxes by moving the slider up and down starting from the box that contains the whole application, to the atomic elements that describe individual user interface widgets. Following such an abstract-to-detail layout modeling, the designer is supported to start specifying statements on the highest abstraction level possible. The editor visualizes atomic elements blue and boxes that contain nested elements through a yellow overlay like depicted in figure 4.15.

To prevent specifying conflicting statements the designer can only specify relational statements between elements on the same nesting level (which corresponds to the abstraction level of the task tree if the task model has been used to derive the containment). In the editor we use the red corners to illustrate elements that are located on the same nesting level and thus can be target of a relational statement. For instance in figure 4.15 two boxes are visualized for an exemplary application by the red corners that are not directly related: The upper one signals the two boxes "showCurrentStepDetails" and "Help" whereas the lower one consists of one box "stepNavigation" and one individual element "stepSelection". In this case the designer has the option to define an interpretation for the relation between "showCurrentStepDetails" and "Help" but not the option to specify a direct relation containing elements of the upper and the lower box (since such a relation has to be set on a higher level of abstraction which contains both boxes).

Each statement that has been defined is written into the layout model and gets instantly evaluated to a set of constraints that is solved to update the box-based preview. This process happens without any remarkable delay so that we can recalculate the constraints on the fly to give an instant visual feedback.

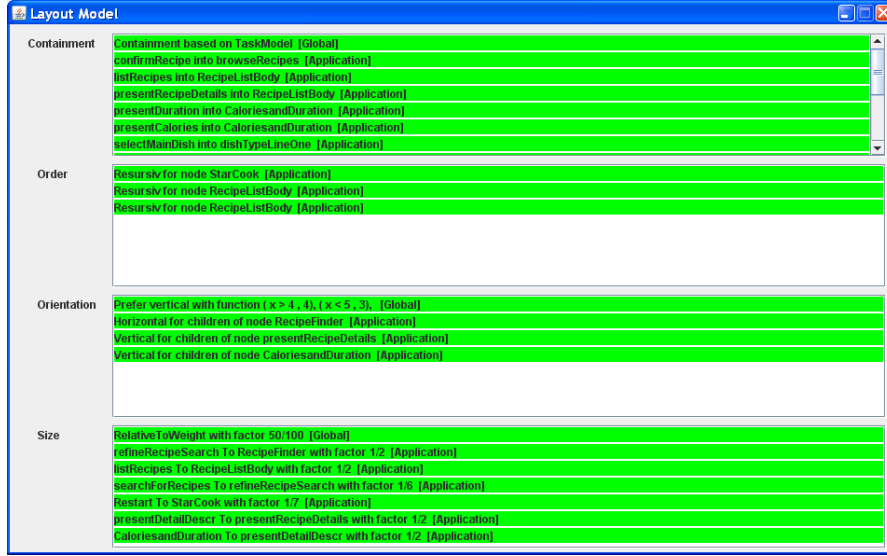


Figure 4.16: The actual layout model consisting of a set of statements that are grouped by the layout characteristic they are targeting to.

Figure 4.16 presents a screenshot of the editor's view of the layout model. The layout statements are grouped by the layout characteristic they are primarily addressing. In case of conflicting constraint sets, the last statement that the designer has entered and that caused the conflict is highlighted red.

Since the four layout characteristics cannot be handled independently from each other (for instance the containment constraints the order, orientation and size and the order constraints the orientation and the element size), we define a general order in which the statements are processed based on the layout characteristic they are mainly addressing: Like depicted by the screenshot in figure 4.16 the containment-related then the order-related, after that the orientation-related and finally the size related statements are processed.

After a suitable set of constraint-generating functions have been identified, the designer can check the resulting layout for its adaptivity to manage certain context-of-use scenarios by browsing through a set of predefined contexts-of-use. Predefined contexts-of-use contain further context-specific layout statements that have been specified independently from a certain application and are reflecting the capabilities of a device or the preferences of a user. Like illustrated in figure 4.12 the layout statements of predefined contexts-of-use are merged to the layout statements of the application to simulate the user interface layout.

In parallel to the layout model derivation the concrete user interface modeling can be done. The following section describes the activities of this design step in detail.

4.2.6 Concrete User Interface Modeling

The concrete user interface (CUI) modeling sub-phase is about the derivation of platform specific presentations based on the task, AUI and domain model specifications and consists of two main activities: First, the execution of a set of generic transformations that construct at least one basic CUI transformation for each platform that should be supported. Second, the design activity, in which the basic CUI transformations are enhanced to form final CUI transformations. The final CUI transformation is stored within the CUI model and is executed at run-time for each interaction task that should be presented as part of a user interface on a certain platform.

During the CUI design process previews can be generated based on the basic CUI as well as the final CUI transformations. Therefore the exemplary application objects generated during domain modeling as well as the presentation tasks that have been generated during the task modeling can be utilized to support a realistic final user interface preview. Figure 4.17 illustrates the process of the CUI modeling sub-phase.

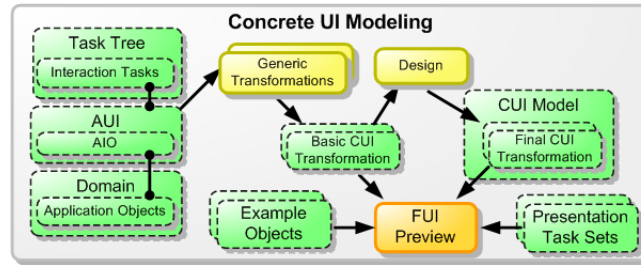


Figure 4.17: The CUI modeling step transforms the AUI to UI by applying transformations.

In the next sub-sections we describe the process for the automated multi-level transformation in greater detail, present the CUI model notation that utilizes the extensible stylesheet language transformations (XSLT), discuss the testing opportunities of this sub-phase and finally present the tool support for the CUI modeling in the MASP Builder.

4.2.6.1 Process: Automated multi-level transformation

Similar to approaches like UsiXML and TERESA we rely on transformations to produce the final user interface. But instead of producing all final user interfaces for all platforms during design-time, we shift the final user interface generation to run-time, which offers the advantage that we can more flexibly adapt the user interface to meet the actual context-of-use. The main drawback of the run-time generation is the performance of the transformation. Therefore we follow a multi-level transformation approach, which starts with a generic transformation that interprets the task, domain, and AUI models at design-time to produce a more specific transformation that is suitable to be processed at run-time. The generic transformation implements a general and automated interpretation of the other design models for a specific platform. Target platforms include an HTML-based web-browser, a VoiceXML-based speech recognition and synthesis

system, and even technologies like for instance templating engines or a XUL interpreter. The generic transformation produces a specific transformation that can be targeted to (manual) improvements and implementation of certain design decisions.

Since the generic transformation is predefined and automatically applied at design-time, it can be designed in a way, that the resulting specific transformations:

- all follow the same structure so that the designer does not need to spend time getting used to additional transformations for new platforms.
- already contain most of the final user interface source code, so that the designer has not to be a specialist knowing all the technical details for implementing a user interface on a certain platform, but can concentrate on the actual look and feel instead of spending time on “getting things to work”.
- are optimized to be efficiently interpreted at run-time. Different to humans, which like to produce clear and human-readable source code, automated transformations can be optimized to be efficiently machine-processable.
- ensure a separation of concerns and therefore support re-usability.

4.2.6.2 Notation

We implemented the multi-level transformation approach by using XSL transformations, which offer the benefit that both the generic as well as the specific transformations can be implemented in the same declarative XML-based language.

Each specific template follows a fixed structure, starting with a prologue containing the production of platform specific initializations. For instance when generating a HTML presentation, the prologue includes the page header including page title, the cascading style sheets paths, java script functionalities realizing form checks or presentation effects and variables to store the user session. After the prologue the presentation structure is produced, which implements the structure as specified in the AUI model. The last section contains the production statements for each AIO consisting of the selected domain object data and the additional information as added during the AUI enrichment activity. By following a separation of concerns into this three sections, the manual design activity is typically only required to be done once for each platform. Firstly for implementing the applications’ look and feel by adding a cascading style sheet, and secondly for each AIO that can be often reused if the same application object is re-ocurrently presented in several screens (or applications).

Listing 1 Excerpt of an specific AIO to CUI transformation.

```

1 <xsl:template match="frame[@type='de.dailab.shs.ctrl.ontology.
2     SHS_Device:DAI_1.Clock']">
3   <div id="de.dailab.shs.ctrl.ontology.SHS_Device:DAI_1.Clock">
4     <tr>
5       <td></td>
6       <td style="font-weight:bold; font-size:11; padding-left:25px">
7         <xsl:value-of select="attribute[@select='hour']"/> :
8         <xsl:value-of select="attribute[@select='minute']"/>
9       </td>
10    </tr>
11  </div>
12</xsl:template>

```

Listing 1 shows an exemplary AIO production statement that has been generated by a generic HTML transformation to produce a presentation of a clock with the actual time as part of the user interface. The automatically generated part includes the matching condition of line 1, which states, that an application object that instantiates the clock category of the “SHS_Device” ontology should be rendered to HTML as part of a container (the “div” tag of line 3). From the clock category the attributes hour and minute should be selected and presented to the user separated by a colon. The specific part that has been added by the designer is shown in line 4-6 and 9-10. In this lines styling and design information are added consisting of a static clock picture (line 5) and font styling information (line 6).

Listing 2 Excerpt of an generic AIO to CUI transformation for generating a specific transformation for a HTML platform.

```

1 <aidlnew:template match="attribute">
2   <aidlnew:choose>
3     <aidlnew:when test="not(@input='true')">
4       <xsl:value-of>
5         <aidlnew:attribute name="select">attribute[@select='
6           <aidlnew:value-of select="@select"/>']/label[@lang=$language]/.
7         </aidlnew:attribute>
8       </xsl:value-of>
9       <xsl:value-of>
10        <aidlnew:attribute name="select">attribute[@select='
11          <aidlnew:value-of select="@select"/>'].
12        </aidlnew:attribute>
13      </xsl:value-of>
14    </aidlnew:when>
15    <aidlnew:when test="@input='true' and @type='multichoice'">
16      <br/>
17      <input type="checkbox">
18        <aidlnew:attribute name="name">{attribute[@select='
19          <aidlnew:value-of select="@select"/>']/@id}
20        </aidlnew:attribute>
21        <aidlnew:attribute name="value">true</aidlnew:attribute>
22      </input>
23      <xsl:value-of>
24        <aidlnew:attribute name="select">attribute[@select='
25          <aidlnew:value-of select="@name"/>'].
26        </aidlnew:attribute>
27      </xsl:value-of>
28    </aidlnew:when>
29  </aidlnew:choose>
30 </aidlnew:template>

```

Listing 2 presents a condensed extract of a generic AIO to CUI transformation. The excerpt shows two production rules for handling the AIO selection of an attribute (line 1). The first one (line 3-14) handles the basic attribute selection that should result in a AUI-to-CUI transformation that extracts the attribute values for presenting it in the user interface together with an enriched label text. An example of the resulting AUI-to-CUI fragment of the specific transformation has been shown in listing 1. The second production rule (line 15-28) is used to generate a transformation that selects attributes of a set to form a multi-choice list that will be rendered as a list of check-boxes in HTML (line 17). This production rule is called for each element of the list.

Listing 3 Excerpt of an generic AUI to CUI transformation prologue for a HTML platform.

```

1 <xsl:stylesheet version="1.0">
2 <xsl:output method="html"/>
3 <xsl:param name="language" select="'de'"/>
4 <xsl:template match="scenario">
5   <aidlnew:choose>
6     <aidlnew:when test="//list[@input='true']|//attribute[@input='true'] ">
7       <form method="POST">
8         <aidlnew:attribute name="action"{$locationParameter}</aidlnew:attribute>
9         <input name="sessionid" type="HIDDEN">
10          <aidlnew:attribute name="value"{$sessionId}</aidlnew:attribute>
11        </input>
12        <aidlnew:apply-templates select="./container" mode="include"/>
13        <br/><br/>
14        <input type="submit"></input>
15      </form>
16    </aidlnew:when>
17    <aidlnew:otherwise>
18      <aidlnew:apply-templates select="./container" mode="include"/>
19    </aidlnew:otherwise>
20  </aidlnew:choose>
21 </xsl:template>

```

Listing 3 presents an excerpt of the prologue of a generic AUI to CUI transformation. Beneath defining the default language (line 3) the HTML transformation prologue checks for AIO elements that have been specified as input elements (line 6). If input elements are part of the user interface, the generic transformation produces a form as part of the prologue (line 7) before traversing through the AUI container hierarchy (line 12 and line 18). The AUI containment structure forms the CUI structure. Therefore the generic transformation ensures that all AUI containers are traversed as specified in the AUI model hierarchy. For each container design decisions can be added in the specific transformation similar to the AIO production statements that have been presented in listing 1.

4.2.6.3 Testing Step

Like illustrated in figure 4.17 the CUI model testing step is about producing instant previews of the final user interface for all platforms that should be considered for the interactive application. The previews can be generated on the one hand by the generic transformation, which produces a technical, but well structured user interface without considering any specific user interface design. On the other hand, as soon as the specific transformation has been enhanced, it can be applied iteratively to check the final user interface design. Both previews take advantage of the exemplary application objects that have been defined during domain modeling and the presentation task sets (PTS) that have been specified during task modeling. Despite the fact that the FUI previews

only offer user interface presentations that are based on static content, the exemplary objects enable the designer to get a clue about the amount and type of information that is presented as part of the user interfaces. By using the exemplary objects all parts of the user interface that strongly depend on data that is created during run-time can be previewed in a realistic situation of the final application. With the help of the PTS not only final user interface fragments can be previewed but also complete screens involving several tasks can be checked for a consistent screen layout design.

4.2.6.4 Tool-support

To support the CUI modeling process, we have implemented an editing and previewing functionality as part of the MASP Builder tool.

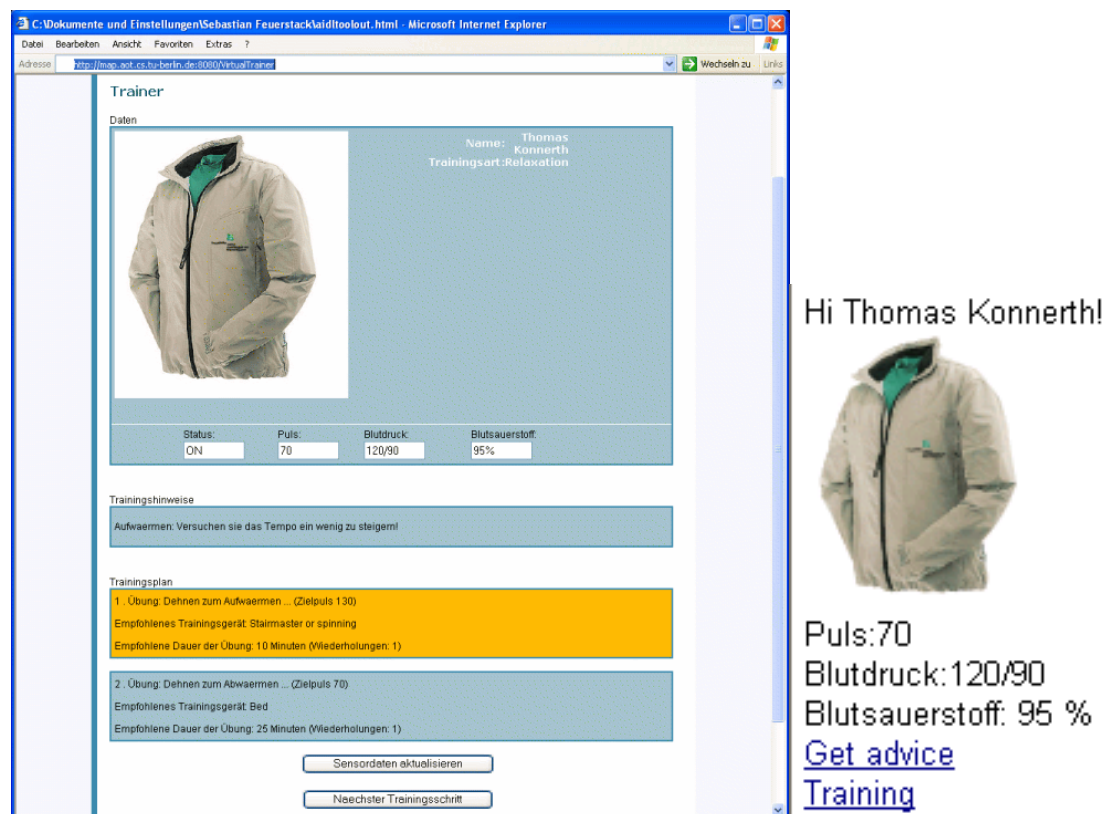


Figure 4.18: Final user interface previews for HTML using a webbrowser (left) and WML using a cell phone emulator (right) based on exemplary application objects for a certain PTS.

Figure 4.18 presents two previews of a work-out support application, which can be accessed before and after the work-out via a web-browser on a desktop pc, and during the workout by using a smart phone (WML) or a headset by using a speech command interface. Both previews have been generated just by applying generic transformations

for all platforms and consider exemplary application objects containing the actual sensor data, the training advices, and a certain PTS. Thus the user interface does not include aesthetical design improvements but still gives an impression about the user interface controls and data for all platforms.

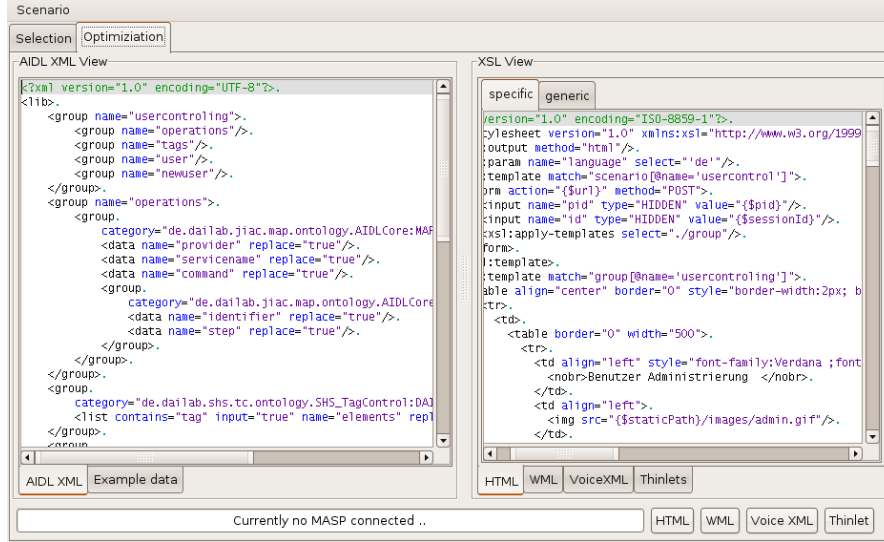


Figure 4.19: Screenshot of the XSL transformation editor of the MASP Builder. On the left side, the actual abstract user interface is presented in an XML-based format. On the right, the designer can switch between several XSL editors to manipulate the AUI-to-CUI transformations.

Figure 4.19 illustrates the XSL editing functionality of the MASP Builder. The tool uses a two-column layout which presents the complete AUI for a certain PTS with the included exemplary application objects (which we call a “scenario”). Several of such scenarios can be configured in the MASP-Builder and can be easily browsed by using the pull down box under the left column. In the right column a platform specific XSL transformation is visualized. Initially this transformation can be automatically generated by using the generic AUI-to-CUI transformation (which can be accessed by a pull-down menu that includes additional options to automatically identify and validate the specific XSL transformations). At the bottom of the right column, the designer can switch between several specific transformations (currently we support WML, HTML, VoiceXML, and Thinlets) by using the tabs as well as applying the transformation to generate an instant preview. The previews are presented by using end device emulators (for instance to preview the final user interface on a cell phones or on a web-browser).

Port to listen for AIDL scenarios

Socket port: 5555

HTML

Generic XSL: de/dailab/jiac/map/gui/genericHTML.aidl.xsl Browse

HTML Bro... firefox Browse

WML

Generic XSL: de/dailab/jiac/map/gui/genericWML.aidl.xsl Browse

WML Brow... wmlexplorer Browse

Thinlet

Generic XSL: de/dailab/jiac/map/gui/genericTHINLET.aidl.xsl Browse

Thinlet Browser: firefox Browse

VoiceXML

Browse Generic XSL: /dailab/jiac/map/gui/genericVXML.aidl.xsl

☒ VoiceXML Server

Server: voiceserver.sercho.de

Phone: 31474060

User: serchouser

Password: *****

☐ VoiceXML Editor Browse

Classpath

Name	Path
shs.jar	/media/...

Remove JAR Add JAR

Cancel Save Apply

Figure 4.20: The configuration dialog of the MASP builder tool supports configuring several device emulators as well as connecting to a Voice Server and a running MASP environment to directly manipulate a running system.

Figure 4.20 present the configuration dialogue of the MASP Builder, wich allows to configure the generic AUI-to-CUI transformations and to specify the browsers and end device emulators that should be used to display the preview. The MASP Builder can be directly connected to a VoiceXML-based voice synthesis and recognition server. Therefore the server location as well as a phone number can be configured by the designer. In this way a user interface preview can be generated for the voice command interface as well. As soon as the designer presses the generate preview button for presenting the voice interface, he receives a call for the configured telephone number and can preview the speech interface.

Although the user interface is already technically usable after the specific transformation has been generated, the design often requires some re-work (user interface design is still an art!), which includes adding a specific style (we use CSS style sheets) and must be manually done by enhancing the specific transformations. Each modification can instantly be checked by requesting a preview of the user interface based on the given example objects and the selected PTS. The result of the CUI modeling is a platform specific transformation (that works on the abstract user interface description) and includes specific design definitions.

As we used our tool successfully in numerous projects we experienced the need for

supporting reusability mechanisms that allow the extraction of templates from specific transformations. At present this has to be done manually in the user interface construction process. Each template that is extracted can be described by a new reference and is stored in a library that is used by the automated generic transformation. The developer can refer to these types in the context enrichment during the AUI modeling.

Performance considerations The structure of the specific transformations is optimized regarding typical performance issues since it takes special care of the matching conditions that can significantly reduce performance if the internal transforming functionality is unknown to the developer. In listing 1 one can observe that we always use a direct template selection in the xpath-based matching condition, as this kind of selection provides the best performance.

4.2.6.5 Service Model

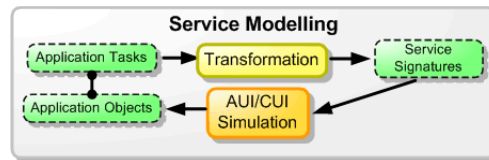


Figure 4.21: The Service creation step of the implementation phase creates service signatures based on the application task's information.

Finally the last sub-phase creates the service model by applying a transformation to all application tasks and their corresponding application objects to create service signatures containing the input application objects as parameters and the output application objects as service results.

4.2.7 Deployment and Implementation phase

The deployment phase requires a run-time system as it will be described in chapter 5.2.1, which offers Model-Agents conforming to the same meta-model that is used to describe the models designed by the methodology. Each non-transient model that has been developed during the development process is deployed together with the mappings between the models into a run-time environment that executes the application. The deployment phase therefore is mainly concerned with configuring each Model Agent with the model it needs to process at run-time. Deploying all developed models and starting the system creates an instance of all the models in the run-time environment that defines complete the state of the application.

4.3 Conclusions

We presented a tool-supported method for interactive system development that explicitly considers testing support in all other phases and steps of the development process. Different to other approaches the method covers all phases of the development lifecycle and is directly related to a run-time architecture that will be presented in the next chapter. Whereas the concepts and tools of the method can be used independently from the run-time architecture, coupling both enables to bridge the gap between design-time and run-time of interactive system development. Most of the design-time models that are modeled as part of the method can be directly executed in the run-time environment.

Figure 4.2 illustrates the sequence of tools that support the method. Whereas in the early analysis phase and the domain modeling step of the design phase we rely on already existing tools (the EL-Tool [Paternò and Mancini, 1999] and the JIAC ontology builder [Tuguldur et al., 2008] respectively) we introduce several new tools to support the designer in all phases of the method. The method has a strong focus on testing support to uncover misunderstandings and misinterpretations as early and often as possible. Therefore all tools support generating previews, enable model checking or offer to run simulations enabling the developer to test intermediary results of the development process. We described the method by following a fixed structure for each step of the development process: After an introduction we explained the activities required to perform each individual step of the method. Thereafter we introduced possible notations that can be considered for specifying the models. Then we described the testing and tool support for an efficient development.

Different to other approaches the application can be comfortably maintained and extended even after the initial development has been undertaken. This is possible since all design models are kept alive in the run-time environment and can be targeted to manipulations on all levels of abstractions that are supported by the models. Both the MASP Builder that supports the AUI and CUI modeling as well as the Layout Generator can be directly connected to the MASP run-time environment in order to manipulate and update the models to consider changing requirements. Bugs can be uncovered by the MASP debugger enabling the designer to inspect concrete model instances and manipulate states.

5 The MASP Software Model for Interactive Systems

Whereas model-based approaches introduce various abstractions to design interactive systems, each enabling a comprehensive view of the whole application, they are currently focused on analysis- and design support. Thus, these approaches require the developer to consider all combinations of devices, which the interactive system should be able to adapt to during the development process. Each new mix of devices and modalities requires going through most of these design models again to compile a new user interface.

Realizing such an approach requires two problems to be solved:

- A notation that is comprehensive enough to be interpreted at run-time while also providing the application designer with an overview of the whole application, because adding new multimodal interactions often affects the complete interaction flow of the application. The notation should support prototyping of new interactions without changing the original application and should be supported by tools.
- The run-time environment must reflect commonly used design-time models describing several abstract views of an interactive application and has to support mappings to the run-time environment to eliminate the gap between design and run-time. Every modification to the models that is done during prototyping has to be instantly propagated to be synchronized with all other models.

In the next section we systematically derive the requirements for an model-based architecture for an interactive system and then present our approach for a MASP software model for interactive systems in section 5.2.

5.1 Requirements Derivation

5.1.1 Architectural Observations

Run-time environment vs. Development environment

Actual approaches either focus on implementing a run-time environment or an integrated development environment. Run-time environments are proposed to offer adaptation mechanisms to the actual context-of-use. Therefore these approaches consider the capabilities of the target platforms, the user's preferences, and the surrounding environment where the interaction takes place. Calvary et al. [Coutaz and Thevenin, 1999] suggested the plasticity property to describe an adaptation to different contexts of use while

preserving the user's needs and abilities. To support plasticity they introduced comets [Calvary et al., 2004], a special kind of widgets that publish their guaranteed quality in use, the tasks they support and the domain concepts they consider, and can self-adapt to different contexts-of-use.

Coninx et al. [Coninx et al., 2003] introduced the Dygimes framework, a model-based approach that is able to combine several models at run-time to generate user interfaces. They use task specifications that are associated with user interface building blocks to generate interfaces that can adapt to the context-of-use. [Klug and Kangasharju, 2005] directly execute a task model at run-time and support the generation of interactive systems that are able to adapt to the actions of the users by allowing manipulation of the task structure at run-time.

Integrated development environments assist the user interface designer specifying the different models reflecting different views of an interactive system and various levels of abstraction. In TERESA [Mori et al., 2003], the designer has to start with a task analysis to identify the basic processes that should be supported by the final application. In the task modeling phase temporal operators are used to relate tasks to each others and objects are associated to task that are required for successful task performance. Internal heuristics of TERESA help the designer to derive a dialog model that is basically constructed of enabled task sets building the foundation for the generation of the final presentation. Vanderdonckt et al. have implemented a lot of different tools, all loosely coupled together using UsiXML [Limbourg et al., 2004a] as the modeling language to describe the diverse models of abstractions that can be constructed or manipulated by the designer. They propose specialized tools for specific audiences to cover various development approaches. For prototyping approaches different levels of fidelity are supported: SketchiXML [Coyette and Limbourg, 2006], a sketching editor is targeted to low-fidelity prototyping and replaces paper sketching of user interfaces, the VisiXML tool [Vanderdonckt, 2005] supports vector drawing, whereas GrafiXML [Limbourg et al., 2004b] is an advanced user interface editor supporting high-fidelity prototyping.

5.1.2 Architectural Shortcomings

Limited support for run-time adaptation to user and platform

Different to early approaches like Mastermind [Szekely et al., 1995], actual approaches concentrate on architectures implementing tool-driven environments to support the user interface designer. Run-time environments are faced with the additional challenge to support platforms or devices that are unknown to the designer during the analysis and design phase. Further challenges include performance considerations to guarantee response times that do not restrict the user during his task performance. UIML is a prominent representative where user interface generation often happens at run-time. Therefore user interface *renderers* are deployed on the user's end-device that are able to interpret the user interface language and dynamically generate a presentation [Schaefer et al., 2004, Schaefer and Bleul, 2006]. Unlike earlier works that dis-

cussed transcoding mechanisms to dynamically adapt a user interface to different platforms, actual research concentrates on layouting mechanisms as one suitable option for realizing user interface adaptation at run-time. Early works by Alan Borning [Borning et al., 1987] have been continued by [Myers et al., 1997] and recently by [Luyten et al., 2003b, Luyten et al., 2006b] to support multi-device layouting based on the cassowary algorithm proposed by [Badros et al., 2001].

In actual model-based approaches, domain models are generally accepted to serve as an input for the automated user interface generation on a conceptual level. The upcoming semantic web offers standardized and machine readable descriptions like for instance RDF and OWL that can complement the domain model information by automated extraction at system run-time, which currently has to be specified manually at design-time. A first attempt to consider the information offered by semantic services during the user interface generation process at run-time has been proposed by [Kaltz et al., 2005].

On the task model level that abstracts from any modality or platform, adaptations of the task structure to support different platform and device capabilities at run-time have actually rarely been considered. Although different approaches exist that execute the task model to run an interactive system, so far only [Klug and Kangasharju, 2005] describes an approach that allows task manipulation at run-time to support adaptation to different users, but did not consider to address device capabilities.

5.1.3 Architectural Requirements

5.1.3.1 Conceptual simplicity

In recent years technology has advanced the development of new kinds of devices useful to enable new interaction possibilities and is moving rapidly towards the vision of the disappearing computer. To support mechanisms offering *modularity* and *extensibility* of the architecture and enabling a seamless integration of new interaction capabilities as soon as they get available, an architecture based on a straight-forward design is required. In order to do so, previous architectures for model-based approaches have to be reconsidered to support a run-time architecture, and a meta-model has to be developed describing the general concepts of a model-based architecture and the required layers to support a straight-forward derivation of a model-based architecture.

5.1.3.2 Separation of concerns

Following the concepts of the reference architectures that have been discussed in chapter 3, a reference model has to be designed that pays attention to a model-based run-time architecture. Based on early and already well accepted models like Arch/Slinky or PAC-Amodeus that propose general reference models for interactive systems, actual requirements regarding device independence by supporting various levels of abstraction have to be taken into account and a refined reference model has to be designed.

5.1.3.3 Performance considerations

Different to model-based proposals that utilize various models to derive a final user interface via a process of model transformation, a run-time architecture has to do all these model-transformation steps instantly during the interaction with the user. Beneath the de-facto standard of measuring the average response time that the architecture requires for the propagation through the models considered in one interaction-cycle with the user, scalability and caching options should be addressed required to serve user interfaces for big audiences accessing an application that is based on the proposed architecture using the internet.

5.1.3.4 Consideration of a methodology

The design and development of an interactive system that is based on a model-based run-time architecture has to be conducted by a methodology guiding the involved roles efficiently. Since a transformal development approach like described recently by [Limbourg et al., 2004b] is an established process to develop multi-platform user interfaces [Paternò, 2005, Calvary et al., 2003, Luyten et al., 2003c], the designed architecture has to explicitly expose the various models reflecting the actual status within the development process. Each model considered by the architecture may be subject to analysis and manipulation during the development steps [Puerta and Eisenstein, 2003].

5.1.3.5 Prototyping support

Recent proposals in model-based development implement multi-step development enabling development into various directions (bottom-up, top-down, wide spreading, etc.) [Calvary et al., 2003]. Entrypoints are used to indicate the potential initial models where development can be started. To support early prototyping the architecture must be flexible enough to rely on the models in the sequence they are used by the developers and designers of the interactive system. Thus, the architecture has to run with every model that is supported to be an initial model (entry point) to development.

5.1.3.6 Support for adaptation at run-time

Only limited support for adaptation mechanisms can be considered at design and development time. As soon as the final user interface is produced, further adaptations are limited to apply transcoding mechanisms or doing a re-iteration through parts of the development cycle. Beneath consideration and adaptation to evolving technologies that have been previously unknown in the development process, dynamically changing situations of the users that are captured by context-of-use models require a more flexible adaptation mechanism that can be done instantly at run-time.

5.1.3.7 Tool support

The productivity of multi-platform development heavily depends on the tools available to support the development process. The architecture has to consider connection points

enabling easy model-manipulation at run-time to support maintenance and debugging, as well as instant testing of changes of a running application.

5.2 The Multi-Access Service Platform

The requirements that need to be considered by user interfaces are manifold. Beneath the interactive application's domain the user interface has to consider multi-modal interaction, needs to be adapted to several contexts-of-use, should be able to migrate to follow the user or even might be distributed to enable situated computing. Two decades of model-based user interface development have resulted in several promising approaches to address these requirements. Most approaches have been developed isolated from each others or have been focused on a specific problem domain. Only little work has been done so far to concentrate the efforts to a standardized approach like proposed by the Camelot Framework [Balme et al., 2004] or by specifying the UsiXML format [Limbourg et al., 2004b]. These approaches end up with a very-large model-based development process for developing user interfaces [Vanderdonckt and Berquin, 1999] requiring a tool-chain and a set of models considering all the requirements that have been listed above. With each model specifying a certain abstract view on an interactive application or describing a specific requirement like for instance the supported context-of-use adaptations, the development complexity gets increased: Each user interface model needs to be comprehensively understood by the designer to be utilized accurately. Moreover, with each introduction of an additional model new model-to-model mappings and transformations need to get included into the model-based development process. These mappings are very hard to observe and to maintain by a designer especially if one model is connected to several other models.

Because of this and the big initial learning efforts to identify and capture effective abstractions into models, so far the model-based development of interactive systems has been applied by the industry to a very limited extend. Applications are still implemented following very basic design patterns such as the model-view-controller (MVC) pattern that is an integral aspect of actual modern frameworks like for instance Ruby on Rails and Java Server Faces¹. Whereas MVC offers a separation of the application's model from the presentation, MVC-based application designs results in some models, each accessed by several controllers and each controller accessed by multiple views. Thus, implementing large interactive applications ends up with a modularization of model/view/controller modules making a comprehensive view on the entire application at implementation time impossible. Task and Concepts modeling are often considered as paper-work and are not directly connected to the actual implementation, which complicates later enhancements of the application to support new contexts-of-use. To overcome these problems we propose the Multi-Access Service Platform (MASP), which implements a modular agent-based architecture and can be flexibly composed of those user interface models that the designer feels most comfortable with and considers as most effective to support his work.

¹JSR 252: JavaServer Faces 1.2; SUN Microsystems; Online at: <http://jcp.org/aboutJava/communityprocess/final/jsr252/index.html>; Last Checked: 02/01/08

This chapter introduces our approach to utilize user interface models as basis for a run-time system allowing the derivation of flexible user interfaces from the defined models. Both, the run-time system as well as the models are described by a meta-model, ensuring consistency across all models at run-time and design-time. The approach aims for the integration of the run-time and design-time view of the user interface to achieve greater flexibility and adaptability of user interfaces for smart environments. The adaptation of user interfaces to various environments unknown at design-time as well as context information and usage situations is supported. This integration is achieved through the combination of a run-time system capable of interpreting user interface models with a methodology allowing the step by step creation of the required models.

We start with an introduction of the relationships between models and instances and point out how this view to run-time models can be utilized to generate user interfaces through the interpretation of a user interface model at run-time in the next section.

5.2.1 Run-time Interpretation of User Interface Models

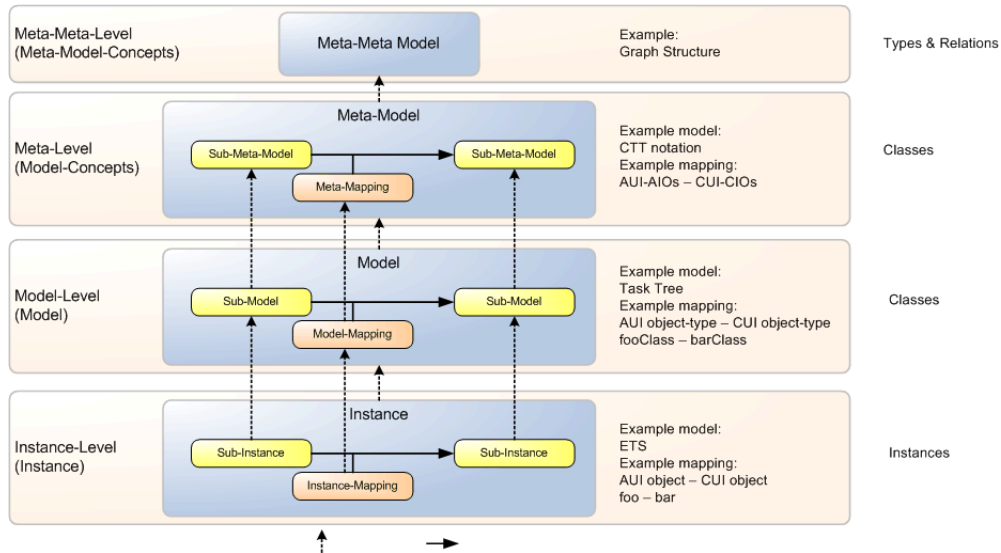


Figure 5.1: (Meta-) Levels of UI Models.

Figure 5.1 illustrates the relations between the different levels of abstraction underlying our approach to create models that allow the derivation of user interfaces at run-time. Each level of abstraction comprises a model composed of a number of sub-models as well as relations between the sub-models. Each sub-model describes part of the model to provide a consistent, self-contained view to an aspect of the user interface and instantiates the sub-meta-model of the corresponding higher level. In addition, mappings describe the relations between the different sub-models and the mappings between the sub-meta-models. The relations between different levels of abstraction allow the clear separation of system design-time - working on the meta-level, application design-time - working on

the model-level and run-time - working on the instance-level.

At system design-time the developer defines the meta-model and also adopts or creates a meta-meta-model. While the meta-meta-model defines the concepts of the meta-model, the meta-model itself specifies the user interface description language used to describe the models. The CTT notation or an abstract user interface description language are sub-meta-models on this level. Meta-mappings define that Abstract Interaction Objects (AIOs) of the Abstract User Interface (AUI) are related to Concrete Interaction Objects (CIOs) of the Concrete User Interface (CUI) for instance. At application design-time, the user interface model, instantiating the meta-model is composed from different sub-models, instantiating the sub-meta-models. A Concurrent Task Tree defining an interactive application is a sub-model instantiating the CTT notation for example. Mapping an AIO (of type `fooClass`) to a CIO (of type `barClass`) is a model-mapping at this level. At run-time, the created models are instantiated to describe the state of a model at a specific moment in time. An Enabled Task Set (ETS) declaring several tasks as active is an instance of the CTT sub-model defining the state of the interactive application. The fact that a specific instance of an AIO (object 'foo' of type `fooClass`) is related to an instance of a CIO (object 'bar' of type `barClass`) is an instance-mapping. At the instance level this results in the CIO being updated as soon as the AIO changes.

Looking at the level of instances of models at run-time requires a concept of loading and instantiating the models that also defines how the instances can be manipulated at run-time while preserving consistency. Our approach to interpret models at run-time - introduced in the next section - is based on the concept of Model-Agents, instantiating the model to provide access to the instance.

5.2.2 The Model Agent Concept (MAC)

PAC and MVC are prominent implementation models for interactive applications. Both decouple business logic from the presentation. But whereas in MVC the controller is designed to handle only user input, in PAC the controller uses interactive objects to map both, user input and output between the abstraction and presentation to shorten the user feedback loop. Both, MVC and PAC require a functional decomposition of the interactive system to be realized: Whereas in MVC a functional decomposition is not explicitly defined, PAC is structured according to layers. It composes interactive objects by PAC agents on a lower granularity. On a macroscopic level there is a chief agent that controls the entire application whereas on the lower levels there are fine-grained agents handling specific services that the user interacts with [Avgeriou and Zdun, 2005].

Although the functional decomposition into a hierarchy of simultaneously running entities is elementary to manage large interactive systems [Hirsch et al., 2008], the overview about the entire application is only possible on the highest level of abstraction. Different to a functional decomposition, a model-based architectural approach offers several models, each implementing a different perspective on the application abstraction while preserving the comprehensive view on the entire application. Therefore we propose the Model-Agent concept that brings these design-time models alive, allowing the developer to decide about suitable abstractions to implement and to manipulate a running appli-

cation on these levels of abstractions later on.

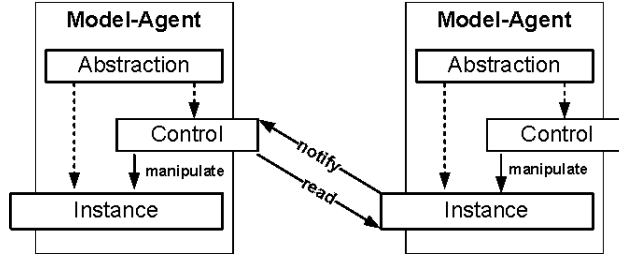


Figure 5.2: Model-Agents are comprised of an Abstraction that defines the agent's behavior (Control) and the structure of the encapsulated Instance. Inter-agent communication is done by the subscribe/notify pattern that is triggered by the controller part of an agent.

Figure 5.2 depicts the basic structure of a Model-Agent that is comprised of three components: The Abstraction that is basically a view (a user interface model) describing both, the concrete interactive application's structure and semantic on a certain level of abstraction. The latter is realized by the Control part that defines the agent's behavior and the latter by the Instance part that contains the agent's knowledge structure for instance about a specific state. The communication between agents is initiated by the observer pattern and always triggered by the Control part that subscribes to another agent's Instance to get notified about updates.

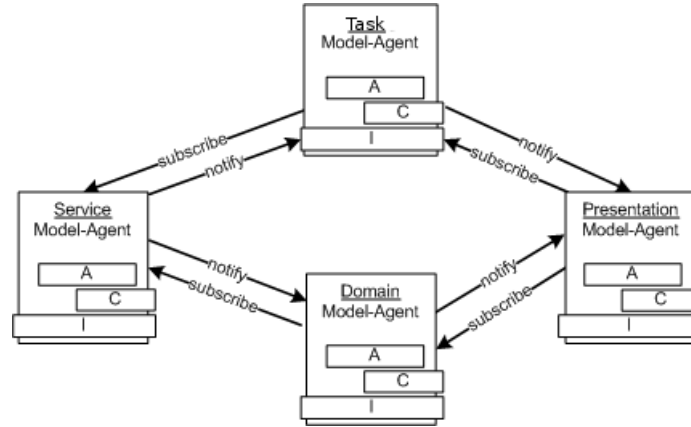


Figure 5.3: At least 4 different types of agents are required to realize an interactive system.

In our reference model, an interactive application is described by at least four different kinds of agents like illustrated by figure 5.3: A Task Model-Agent controls the overall process. Depending whether the task flow contains a system task or an interaction task, the agent delegates the execution to the Service or the Presentation Model-Agent. Results of a service call from the Service Model-Agent get sent to the Domain Model-

Agent whereas the information about a succeeded service call gets propagated back to the Task Model-Agent to continue the task flow. Finally the Presentation Agent registers itself for updates of the Domain Model to be able to update the presentation if domain objects that are part of a presentation get updated.

Following the idea of the Slinky meta-model [Bass et al., 1992] these four agents can be weighted by decomposing them based on the requirements of the interactive application's domain. Thus, a home automation system implementation to integrate several home automation technologies; each represented by a specialized service agent and might include only a basic presentation that is suited for a single touch screen. Whereas a in-car assistive system might include several multimodal-interaction models specifying the various interaction modalities that are supported by the system via a fine grained chain consisting of task, domain, context, concrete user interface Model-Agents as well as an service Model-Agent.

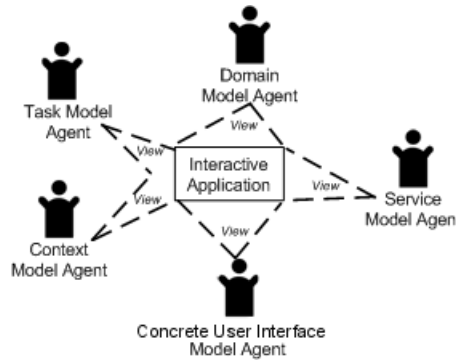


Figure 5.4: Each Model-Agent has a comprehensive view on the entire application on a certain level of abstraction.

Following the Model-Agent concept we identified several advantages:

- *Comprehensive View*; Enhancements can be done easily by modifying just one agent as each agent contains a comprehensive view about the entire application on a certain level of abstraction (figure 5.4). PAC-Amadeus only describes the functional decomposition of the dialog part of an application, and MVC does not offer a guideline on how to relate several MVC parts of a complex application.
- *Minimization of the future effects of changing technology*. New principles and technologies can be introduced by a new Model-Agent containing the additional information or a new interpretation of already available information.
- *Better learning curve*; A developer can start with those abstractions she feels most comfortable with and continuously enhance the application by instantiating new Model-Agents integrating new models to support for instance interface migration and distribution or address the capabilities of new platforms.

- *Integration into a model-based development approach.* Agents can be directly derived from user interface models that have been specified following a model based user interface development process. The agents' communication can be systematically derived from the transformations and mappings between these models.
- *Integration of different user interface description languages.* Each agent acts autonomously on behalf of one model; Only model mappings have to be defined between the interacting agents stating the relations between both models.
- *No gap between design- and run-time of a system.* Model-Agents are directly derived from the design model and each designed model is kept alive and can be manipulated at run-time.
- *No single point of control.* Different to PAC-Amodeus there is no hierarchy. Different views about an interactive application are realized by separate autonomous agents. Each agent can initiate the control of the application based on the internal model information of the agent.

A Model-Agent encapsulates a sub-model and provides access to the current instance of this model. It also encapsulates a processing unit allowing calculations based on the current instance.

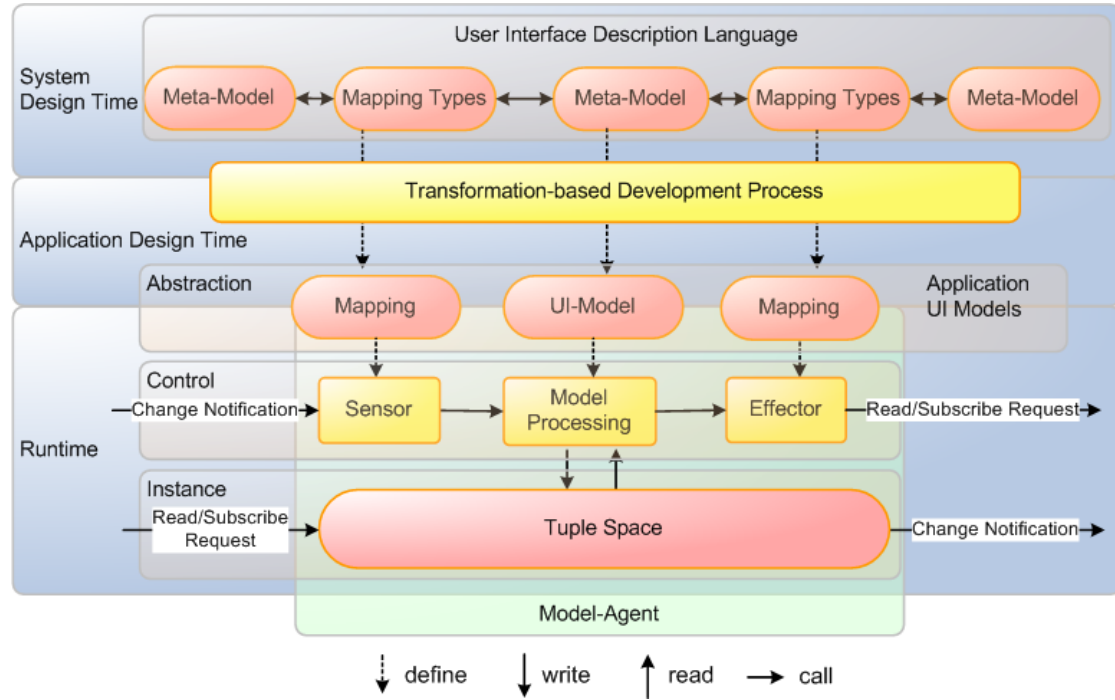


Figure 5.5: For each user interface model one Model-Agent is derived and instantiated through a model-based development process by the MASP.

Figure 5.5 illustrates the components of a Model-Agent and also shows the relations between the system design-time, the application design-time and the run-time of the system. As one can see in the top section of the picture, the Model-Agent as well as the model is defined by the sub-meta-model and the meta-mappings relating the different sub-meta-models. This refers to the fact that the implementation of the agent has to be compliant to the meta-model to ensure that it is able to process all correct instances of the meta-model. The meta-mappings, defining the relations between different sub-meta-models, influence the implementation of the communication capabilities of the agent as they define the set of related agents. This means the developer of the Model-Agent implements the relationships between different Model-Agents based on the meta-mappings, defining i.e. which agents need to communicate and thus have to agree on a communication mechanism and be aware of each other. In addition to the communication capabilities each agent also provides processing capabilities allowing to instantiate and process a model, which puts the agent in the situation of being able to autonomously perform calculations based on the current state of the model - the instance - and the implemented processing rules.

Using the processing and communication capabilities provided, the agent is able to instantiate the sub-model and the related model-mappings. While the model defines the instances the agent is working on at run-time, the mappings define additional, application-specific relations between models, i.e. which objects are related to each other. After loading and instantiating the UI model the agent is initialized and capable of receiving input events from other Model-Agents, encapsulating related models. Received events are processed and can change the model instance, which also triggers new outgoing events. We distinguish between asymmetric and symmetric communication capabilities of a Model-Agent. The former is realized using event-based communication to allow one agent to inform another agent about changes of the model; whereas the latter uses read requests to allow the processing part of one agent to explicitly request state information from a related agent. Events are issued by the notification component, monitoring changes to the instance. Whenever a related mapping is found for a change event the occurred event is propagated to the concerned agents and received by their observer components as defined by the mapping. Read requests are issued through the connector component to the control component of a related agent, when the processing unit requires additional information from a related model. Using only events and read request ensures that each agent can autonomously decide which information is incorporated in the managed model, meaning no agent can write to a model of another agent.

The various agents managing the defined models form an interactive system. With respect to specific requirements of this interactive system, a setting of model-agents has to be identified and mappings to connect these agents have to be specified. One possible setting of an interactive system allowing the distribution of user interfaces to different platforms and modalities is described in the next section.

5.2.3 A Model-based Run-time System

This section describes an approach to realize a Run-time System (RTS) allowing the interpretation of user interface models based on the concepts introduced above. The run-time system loads and interprets the model, manages instances and guarantees that only valid instances are created. It further ensures that mappings between instances are maintained; meaning that updating one instance also affects related instances. The run-time system is structured by the meta-model as the meta-model defines the set of models that can be processed by the RTS. Realizing the run-time system based on loosely coupled Model-Agents provides a flexible and extensible infrastructure that can be adapted to changing requirements or new changes of the underlying models.

Following the idea of using software agents to coordinate the user interface management system [Calvary et al., 2003] we are using an agent-based run-time environment, the Multi-Access Service Platform (MASP) to generate and adapt user interfaces. But instead of requiring a hierarchical organization to several agents like proposed by PAC-Amodeus [Nigay and Coutaz, 1997a], the communication flow between the agents in our environment can be flexibly configured based on the requirements of the interactive application.

As illustrated by figure 5.6 the environment is driven by several Model-Agents where each interprets one user interface model. Different to other approaches [Calvary et al., 2003, Vanderdonckt and Berquin, 1999] that refine a user interface model at design-time to end up with a compiled version of the user interface, we keep all of the models alive at run-time. This allows us to more flexibly react to context-of-use changes that have not been desired at design-time by specifying the required adaptation on an abstract model-level. Each agent is comprised of two parts: a tuple space to store the instantiated model information and a manager containing the semantics and functionality to manipulate the model information. Whereas the manager has complete access to its own tuple space it is not aware of the other agents connected to the system. We connect the agents by using tuple space operations (atomic read/manipulate/write) and the eventing system of a tuple space. The eventing system allows a manager to register for changes of another tuple space. Each agent, handling one user interface model is instantiated once to run a single application but is able to handle several sessions for different users that are accessing the same application. The communication processes between all agents are not hard wired but instead configured for each application based on the user interface models that are relevant for the applications domain. Therefore we can add the layouting Model-Agent as an additional component to the system without making any changes to the other agents.

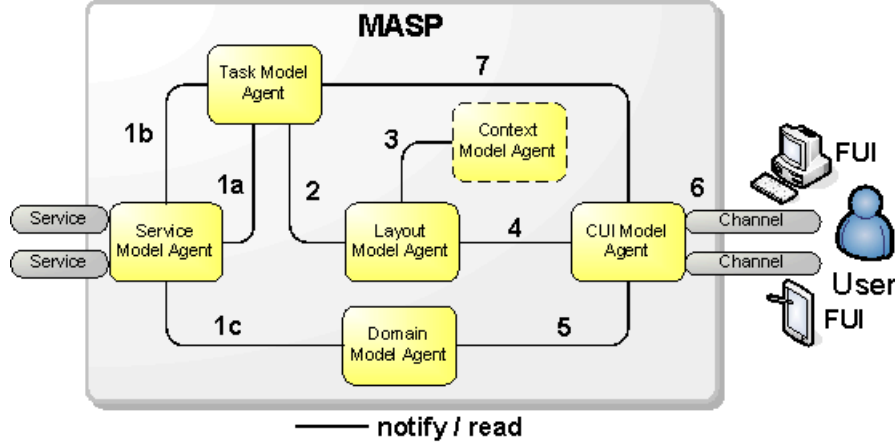


Figure 5.6: MASP Agents Overview.

Figure 5.6 illustrates a possible system of models, allowing the creation of multi-platform user interfaces and briefly sketches the communication between the different Model-Agents. To be able to consistently implement the concept of Model-Agents throughout the whole system, we separated two classes of Model-Agents: Application Model-Agents processing application dependent models developed at design-time by the user interface developer and System Model-Agents processing system dependent models predefined at system development time to define run-time specific behaviour.

During the startup phase of the run-time system each Model-Agent subscribes itself to receive events of the other Model-Agents based on the system configuration. Thus, in the system illustrated by figure 5.6, initially the Service Model Agent registers for changes about the currently active tasks (the enabled tasks set - ETS) with the Task Model-Agent (1a). The Task Model-Agent registers to get notified about on the one hand finished service calls (1b) from the Service Model-Agent and on the other hand to get notified about finished interaction tasks (7) from the CUI Model-Agent. The Layout Model-Agent registers to get notified about each newly calculated ETS from the Task Model Agent (2) and to receive information about context changes from the Context Model-Agent (3). Finally the CUI Model-Agent subscribes to receive new user interface layouts (4) from the Layout Model Agent and to receive changes from the Domain Model-Agent (5) if a service call from the Service Model Agent has resulted in new domain model information (1c) for that the Domain Model agents subscribes itself to the Service Model-Agent. Each updated presentation of the user interface gets delivered by from the CUI Model-Agent by using channels (6) to the different platforms to end up as a final user interface (FUI).

As soon as the run-time system has been successfully started, the generation of the user interface presentation is initiated by the Task Model-Agent that initially loads the task tree that defines the workflow of the application and extracts the initial ETS, defining the current state of the application on the task level. Each task of the tree on the lowest level of abstraction can be either an interaction or an application task. Since the Service Model

Agent has been initially subscribed to receive notifications about each newly activated application task (1a), it starts the relevant services for all enabled services and stores the services' results. For each finished service both, the Task Model-Agent (1b) as well as the Domain Model-Agent (1c) receive a notification. This notification trigger the former one to calculate a new ETS and the latter to stores the service result.

All newly enabled interaction tasks of the Task Model-Agent result in the Layout Model-Agent generating a new layout that includes all currently enabled interaction tasks in one presentation and considers the actual context-of-use, since the Layout Model-Agent is also subscribed to receive notifications about context-of-use changes from the Context Model-Agent. All changes to the domain model (5) or the layout model (4) trigger the CUI Model Agent regenerating the user interface presentation. In case of changes to the domain model the data that is presented in the user interface is updated, whereas changes to the ETS result in adding or removing tasks from the user interface and a new layout respectively. Finally, if the user enters data (6) the domain model will be updated (5) and if she finishes a tasks (7) a new ETS will be calculated by the Task Model-Agent.

Interpreting user interface models at run-time using the described structure allows the flexible distribution of user interfaces to various platforms. Thus, if a new device is connected via a channel to the run-time system by the user, the actual ETS is requested again from the Task Model-Agent and processed as described above to include the new device to present the user interface as well.

5.2.4 Layout Constraint Generation at Run-time

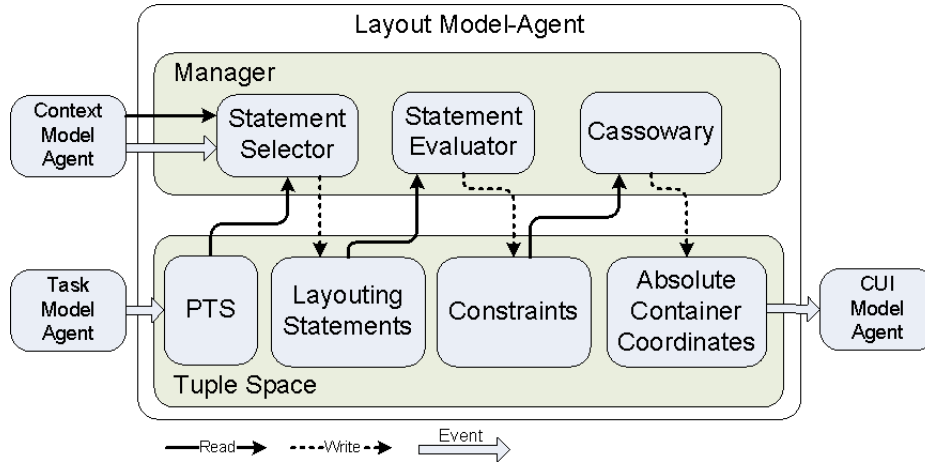


Figure 5.7: Each Model-Agent is comprised of a manager that encapsulates the agent's functionality and a tuple space to store its data.

Figure 5.7 depicts the internal setup of a Layout Model-Agent and its internal as well as its external communication. The agent senses for two external events to happen: First, for a new distribution of the user interface, which has resulted in a new combination of user interface elements to be presented and second a change of the context-of-use. Both

stimulate the agent to select and assemble the layouting statements. The selection of suitable statements is done by the following process:

1. Retrieve layouting statements for the actual context-of-use that have been specified independently from the application and that specify layout requirements to address a certain user or a specific device.
2. Retrieve and select from the ordered statement list of the application layout model the statements for a screen s that:
 - a) address application wide layout interpretation
 - b) address reoccurring elements that are used by s
 - c) directly address the screen s
 - d) are defined for this application and the actual relevant context-of-use scenario.

The statements that have been selected and ordered by priority like listed above are then evaluated to a set of constraints (by the statement evaluator). Thereafter the layout agent finally solves the new constraint setup using the cassowary constraint solver [Badros et al., 2001]. Solving the constraints results in absolute positions for each element of the user interface that are stored within the layouting agent's own tuple space. The CUI Model-Agent is registered for updates to the absolute positions and therefore receives updates for each change of these coordinates that the CUI Model-Agent will use to re-position the user interface elements.

Different to other approaches that use a constraint solver to calculate the user interface layout, we introduced an additional layer of abstraction for defining the user interface layout by a separate layout model that includes statements that are derived using an interactive and tool-supported process and are consistent to the other user-interface models. Since we decide about which statements to evaluate to generate constraints at run-time we can flexibly address layout adaptations to new context-of-use scenarios that can even be independently specified from an application but have been introduced together with a new device or a new kind of user type.

5.3 Conclusions

Realizing the run-time system based on loosely coupled agents enables us to inspect and modify running applications including all instantiated tasks and concepts. Since each agent offers a separate view on the application, tasks, domain objects and service calls they can be changed independently (for instance by removing or detailing tasks) as the agents synchronize these changes automatically making prototyping possible at run-time.

Combining the task model with additional user interface models and mapping these models to a tuple space based run-time system allows the execution of the models supporting prototyping as well as the synchronization of multiple user interfaces distributed across devices and modalities. The notation explicitly includes the annotation of domain concepts to make the task designer aware of concept distribution and allows modeling actions triggered by context changes through the use of application tasks.

5 *The MASP Software Model for Interactive Systems*

The MAC architecture is open for the inclusion of further models depending on the interaction requirements of the application. By adding another Model-Agent to the system that is responsible for layouting of user interfaces based on context-of-use changes at run-time, we demonstrated the extensibility of the architecture.

6 The MASP Abstract Interaction Description Language

Over the last 15 years a lot of User Interface Description Languages have been proposed in a race for the “ultimate” language to describe user interfaces. Such a language consists of a defining syntax to express the characteristics of an interactive system by the terms of the language and a semantic describing the meanings of the terms in the real world. Through the years the goals of such an ultimate user interface description language (UIDL) changed together with the technologies that should be addressed.

The first UIDL’s have been specified in order to modularize an interactive application, separating the user interface from the rest of the application and describing the user interface in a declarative language instead of weaving it into the source code. The separation of the description of the interactive part of an application by a language helped to communicate the specification of the user interface across all involved stakeholders and inversely enabled a user interface construction approach based on a specification of the user interface model [Souchon and Vanderdonckt, 2003].

Together with the introduction and market penetration of mobile phones and personal digital assistants the requirements for UIDLs have been extended to support a wider range of different computing platforms. A lot of new platforms running on appliances introduced their own markup language: WML and cHTML are widely used for cell phones, HTML, DHTML (the interaction of CSS, XSL style sheets, CSS and the Document Object Model, and scripting), XUL, and SwingML are used on traditional desktop PCs. Voice-enabled devices introduced SpeechML, VoiceXML, VoXML and the Java Speech Markup Language. Voice-enabled devices with a display use XHTML+Voice or SALT to realize client-sided multi-modality, which limits the multimodal synchronization to happen on one platform only. With this new markup languages the demand of the users for user interfaces that can be (automatically) adapted to their new end-devices raised. As [Abrams et al., 1999] notes, the explosion in the variety of devices and their associated user interface markup languages has created a Tower of Babel for user interface designers and software developers. This is because interface designers must be familiar with multiple languages, they have to maintain multiple bodies of source code for an application running on different platforms, each requires its user-interface implemented in its own markup language. This situation is very similar to what happened with PCs three decades ago, where many different hardware platforms were developed, each with its own application programming interface.

Its a common practice throughout history to raise the level of abstraction with which people interact with computers. People start with programming computers through mnemonics in assembler after they used strings of zeros and ones. After that their

were compilers that hide the assembler part and enable to program in more abstract constructs with the help of programming languages and additionally enables a wide range of people to program. With the invention of the web and the introduction of the HTML markup language the abstraction level raised again as well as the audience that are now able to implement user interfaces. Thus, it is a common understanding that custom, platform-independent markup languages can help to solve the problem of the Tower of Babel.

In the following section we derive the requirements that should be supported by our approach. Thereafter, in section 6.2 we present language additions to already existing approaches, namely an extended task model (section 6.2.1), a new form of an abstract user interface model (section 6.2.2), and finally a layout model (section 6.2.3).

6.1 Requirements Derivation

6.1.1 Language Observations

6.1.1.1 Diversity of supported abstraction levels and heterogeneity of models

So far a lot of user interface description languages have been proposed, each language considers different aspects or focuses on certain aspects of models. UsiXML [Limbouurg et al., 2004a] is a language specification that covers most of the models that are currently discussed to be included into a model-based user interface development approach. On the task and concept levels UsiXML specifies the task and domain model. The former model is used to derive an abstract user interface model, which is considered as a modality-independent interface specification. For the concrete description level, various models have been proposed to support different platforms like haptic, speech or virtual 3D environments.

Although actual publications in the model-based community seem to agree with this kind of model distribution, they differ in the components that these models contain. Regarding the task model, the CTT approach has been widely accepted as one suitable starting point of a model-based development approach, but the level of granularity that has to be considered to derive an abstract user interface is still an open question. Some approaches like TERESA [Mori et al., 2003] enable a very detailed task tree modeling that allows refinements until the widget level of presentations whereas others switch from a task model to a view on the abstract user interface like IdealXML [Stanciulescu et al., 2005] or derive a dialog model that can be manipulated through dialog graphs like proposed by [Reichard et al., 2004].

The dialog model that is responsible for task-sequencing and navigation has been considered in PAC-Amodeus [Coutaz, 1987b] in the form of the Dialogue Controller, as a central component connecting the functional backend with the presentation of an interactive system. In UsiXML the dialog model is specified as a component of the CUI model [Florins, 2006]. In UIML the dialog part is separately modeled as the behavior part on a more abstract level, whereas XIML specifies the dialogue model separately on a concrete model level.

The models that describe the context-of-use, in accordance with [Calvary et al., 2003] the environment model, the user model, and the platform model, are considered differently in several approaches. Van den Bergh and Coninx [den Bergh and Coninx, 2004a] propose the notation of Contextual ConcurTaskTrees that introduces a new task type called “context task” and therefore consider context information during task modeling. Unlike this approach, [Demeure et al., 2006] propose “comets” to model context adaptation on a widgets level. Comets are “context mouldable widgets that publish their guaranteed quality of use for a set of context-of-uses they are able to self-adapt to” [Demeure et al., 2006]. In UsiXML transformations are used to consider the context-of-use. Transformations can be explicitly defined on each level of abstraction to adapt an interactive system to a different context-of-use.

6.1.1.2 Conceptual closeness

The different proposed UIDLs differ on the one hand in their coverage of the levels of abstraction and on the other hand in the type of models they are able to describe. UsiXML proposed the broadest coverage of concepts and abstractions of all user interface description languages, as it includes concepts to explicitly describe mappings, transformations between the abstractions, and also includes a context model. Thus, the extensive and publicly available UsiXML specification can be a good candidate for a standardized and widely accepted UIDL. Another candidate for a standardize UIDL might be XIML that is currently the only UIDL known to the author that considers a meta-model describing the components of the XIML UIDL. Even though the specification of the concepts of the XIML meta-model are very limited and only consider the specification of a component, an attribute and a relation the XIML meta-model might be a good starting point for an integrated language extension mechanism that is missing in UsiXML. Although UIML allows the definition of new vocabularies to support additional platforms, this approach figured out to be too restrictive and several approaches for “generic UIML vocabularies” have been proposed [Plomp and Mayora-Ibarra, 2002, Schaefer and Bleul, 2006].

6.1.1.3 Focus on graphical modality

The generation of various graphical user interfaces for different platforms is the driving force of actual model-based approaches. Early approaches focused on declarative graphical user interface descriptions that are loosely coupled to the rest of the application to offer better modularity and adaptation of the user interface to new platforms. Nowadays a technological focus on web-based user interfaces can be observed that are targeted to extensions by various technologies like Javascript, DHTML, Java or Ajax. As these technologies are available for all modern web browsers they offer new challenges for model-based user interface generation.

Research on speech-driven user interfaces is often done in delimited research communities. UsiXML offers a set of auditory interaction objects on the concrete (modal-dependent) user interface abstraction level [Limbourg, 2004]. Audio containers represent a logical grouping of other containers or auditory interaction objects (AIO). AIO’s in-

clude auditory outputs like music, voice or simple earcon or auditory input, which is a timeslot for the user to speak something that is recorded and recognized by the interactive system. Despite the fact that UsiXML offers support for voice-based user interfaces, actual tools that are based on UsiXML focus on prototyping or reverse engineering of graphical user interfaces.

First versions of UIML concentrated on a device-independent user interface description of graphical user interfaces. Targets included the user interface generation for WML, HTML and Java platforms through offering different vocabularies. Actual approaches like DISL [Schaefer et al., 2004, Schaefer and Bleul, 2006] or [Plomp and Mayora-Ibarra, 2002] refine the vocabulary definition to support a generic vocabulary consisting of device and modality independent widgets that can be rendered directly on the target device. TERESA [Mori et al., 2003] considers vocal platforms in the task modeling phase and offers transformations from a logical interface composed of interactors for output, interaction and composition to vocal platforms.

6.1.2 Language Shortcomings

6.1.2.1 Lack of generally accepted language

The wide range of different user interface description languages and the absence of a general agreement about the ingredients of such a language exemplifies the range of different aspects that need to be considered when designing a user interface language. Recently UsiXML gets a lot of attention in the research community as it contains several models useful for user interface generation discussed so far. And as it is up to now the only comprehensive language specification that is publicly available, it has all prerequisites to be promoted as a future standardized language. Unlike UIML, UsiXML is more comprehensive by considering the context-of-use and explicitly modeling transformations and mappings. The approaches to combine a language with a basic understanding of the structure of an interactive system, proposed by the *cameleon* unifying reference framework [Calvary et al., 2003, Calvary et al., 2002] might be a good starting point to guide the ongoing discussion towards a generally agreed language and model of an interactive system. Unlike XIML and UIML, which have an industrial background, the UsiXML specification is originated in the research community.

6.1.2.2 Missing extension mechanisms

Another aspect that restricts the progress in finding an agreement about a user interface description language is the absence of clearly defined extension points to enhance a UIDL to consider new technologies, new possibilities and new usage scenarios. Whereas in the software engineering discipline the UML language gained a lot of attention and is widely accepted in recent years one reason for the continuous evolvement of the notation to new technologies and requirements can be found in the detailed description of a comprehensive extension mechanism through a meta-model specification. UsiXML actually misses such a cleanly defined extension mechanism, that XIML in a rudimentary form provides through specifying a minimalistic meta-model. UIML offers a vocabulary as it defines itself as

a meta-language that can be enhanced to support new platforms by the definition of additional vocabularies.

Without a well-defined extension mechanism that is consistent with the underlying models and able to describe all predefined models, each language refinement includes extensive work in the tools that support the language.

6.1.3 Language Requirements

6.1.3.1 Comprehensive lifecycle support

As already stated by Puerta et al. [Puerta, 2001] a user interface description language must enable support functionality throughout the whole lifecycle of a user interface. This includes the representation of language segments for each supported level of abstraction, offering structured storage mechanisms [Puerta, 2001], supporting context models, enabling run-time adaptation and a templating mechanism to enable reusability. Beneath supporting a set of models the language must be able to effectively relate the various elements of various models by supporting mapping and transformation mechanisms like proposed by UsiXML [Limbourg et al., 2004a, Limbourg, 2004].

6.1.3.2 Support for run-time execution

The language has to be able to express user interfaces at a given instant of usage so as to capture relevant information to ensure run-time execution.

6.1.3.3 Adherence to standards

With respect to its compatibility with other systems the language has to consider the use of an underlying technology that is compatible with an industry-based computing model. This requirement is related to the connection to the functional back-end that consists of connecting to various service-based technologies like web-services, messaging systems or special protocols like Jini or Universal Plug and Play. Regarding the design and development the language has to be flexible enough to enable the usage of already existing tools for task analysis and modeling as well as prototyping and reverse engineering, but must not impose any particular methodologies or tools on the design, operation, and evaluation of user interfaces [Puerta, 2001].

6.1.3.4 Well-defined extension mechanisms

As technology continuously evolves and enables new interaction mechanisms between the human and the computer, a user interface description language has to offer an extension mechanism to benefit from new developments. There are already first approaches in recent UIDL's like UIML that offers the definition of vocabularies, or UsiXML that offers model structures that can be extended via sub-typing and its modular ontological framework. Unlike this approaches an explicitly defined meta-model enables compatibility with existing tools if they utilize the language meta-model.

6.1.3.5 Consideration of semantic data

The emerging semantic web offers new opportunities to support the automated generation of multi-platform user interfaces. Since data is described by various kinds of meta-data and refers to ontologies like OWL or Ontoweb, reasoning mechanisms can be used to benefit from these semantic information. A UIDL can integrate semantic descriptions to support the user interface rendering by resolving the internal types of relationships between the data structures that have to be presented or manipulated by the user.

6.2 The MASP Abstract Interaction Description Language

During the design and development of the method and the associated MASP run-time environment we experimented with several user interface description languages. Since it is not our focus to reinvent another user interface description language, we are concentrating on enhancing already existing ones and are only introducing new language models where we missed aspects to complete the language support in order to cover all phases of the method.

We start in the next section by discussing our task model in section 6.2.1, which we identified as the central aspect to realize a user-centered development process. Different to the other language models, a task model, and more specifically the ConcurTaskTree (CTT) notation has gained a lot of attention in the research community. Whereas early approaches focused on the task-based evaluation of interactive applications, more recent research identified the task model as the initial entry for a user-centric interactive application design. Thus, we start the section with identifying actual shortcomings of the CTT notation and continue the section with describing our language additions to realize our requirements listed previously in subsection 6.1.3.

Thereafter in section 6.2.2, we introduce our model to contain the abstract interaction object specifications, which is similar to the UsiXML AUI description, but tightly related to the domain model and the task model from which we directly derive the abstract interaction objects. Finally, we present a completely new model for specifying the layout of a graphical user interface in section 6.2.3. Whereas our abstract user interface model is specified independently from a specific device and modality, the layout model concentrates on deriving graphical user interfaces. Beneath these models further models have been proposed supporting a detailed dialog [Dittmar and Forbrig, 2004], detailed context-of-use [Demeure et al., 2006] adaptations or concrete user interface (UsiXML) modeling, which is out of focus of this work. Compared to other UIDLs, all our models are designed to be directly interpreted at run-time by our MASP environment to close the gap between design and implementation of an interactive application.

6.2.1 Task Tree Model

The run-time interpretation of task models requires a task model notation that specifies the interactive system detailed enough to be executed without further information. Thus it requires integrating static concepts with a temporal description. We found the CTT

notation a good starting point as it allows a user-centric design enabling an abstract to concrete modeling using task trees. The notation also allows focusing on the users' goals by defining sequences of tasks using a LOTUS-based temporal specification. However, for directly interpreting CTT-based task trees at run-time for the creation of user interfaces, we identified several drawbacks of the notation:

- CTT abstracts from the functional core of the system, as for analyzing interactive systems it is focused to the interaction with the user. Further on, the notation does not provide methods to connect sensors and physical control of appliances that should trigger or disable user interaction tasks. Therefore it is not possible to model service backend calls continuously updating the domain model.
- CTT allows specifying objects that are required for task performance but does not provide the possibility to explicitly model the object usage by including it into its graphical notation. From a designer's perspective an explicitly modeled object usage enables consistency and feasibility checks of the concept distribution.
- The notation is mainly targeted to model interactions that are independent from any modality. Modeling modality specific interactions is limited to hide tasks for devices with limited capabilities. Whereas CTT abstracts from how a task is performed by a specific device, decision nodes [Clerckx et al., 2006a] can be used to design device-specific task realizations. To our knowledge, modeling interactions involving the simultaneous usage of more than one device using the task model is not possible by using decision nodes, since decision nodes describe a choice and are not addressing simultaneous usage of more than one modality. Further on interdependencies between the used devices to implement complementary or redundant relations [Nigay and Coutaz, 1997b] between the modalities cannot be addressed.
- The notation also lacks mechanisms to support the reuse of modeled concepts and their provisioning as external trees, which prevents the designer from modularizing interaction patterns for modalities or combinations of modalities.

With a focus on the usage of the defined task tree at run-time, we introduce several changes to the CTT notation. First we describe the four different types of tasks we consider in the notation, after that we discuss the tasks properties that can be used to further specify a task. Then we explain how we connect to the domain model by annotating the concrete objects and their properties to each task that are relevant for the task performance and finally we introduce the temporal operations to specify temporal relations between two tasks.

6.2.1.1 Task Types

Task modeling implements an abstract to concrete design approach. Thus, the designer starts by identifying the overall goal of the user that should be supported by an interactive system. This overall goal is both, complex as it involves several tasks to be reached, and abstract as it needs further refinement to be unambiguously specified and detailed in order

to serve as an interactive application design. Therefore the overall goal is considered as an abstract task and describes the root node of a task tree that needs to be detailed by specifying its children task nodes. Beneath an abstract task we distinguish two basic tasks: application and interaction tasks. An application task is solely executed by the system and does not require any user involvement and does not generate a presentation to the user whereas interaction tasks depend on the user interacting with the system. Compared to the CTT approach that allows application tasks to include a presentation we explicitly separate the functional core from the presentation to the user as modeling the information exchange with the backend system is crucial to be able to use the task tree definition for the user interface creation at run-time. This enables a loose coupling of the functional backend to the user interface and an easier inclusion and manipulation of backend services.

Different to CTT we distinguish two types of interaction tasks: output interaction tasks (OIT) and input interaction tasks (IIT). While OITs require no human intervention but present information to the user until they become disabled by another task, IITs require human intervention like data input, control or selection. Table 6.1 lists the four different task types. Each task type is described by a symbol in the graphical notation of the task tree.





	<i>Abstract tasks</i> specify complex activities that cannot be unambiguously specified.
	An <i>application task</i> is completely executed by the computer without any human intervention through it's internal processing. An application task can generate or manipulate domain objects through it's internal processing. The application task is used to model service calls to the functional core of the application or it can enable other tasks if it is used to model sensor information.
	<i>Output Interaction tasks</i> present information to the user. This could be raw structured data like documents or tables or multimedia output such as video or sound. Output interaction tasks read the objects that are associated with the task once if the are marked as synchronous or until they get disabled by another application tasks or an input interaction task if they are marked as asynchronous.
	<i>Input Interaction tasks</i> describe tasks realizing some interaction techniques to enable the user interacting with the system. Interaction tasks include <i>selecting</i> objects, <i>editing</i> data or enabling the user to <i>control</i> some actions explicitly through events. If an input interaction task is marked as synchronous it has to contain at least one control object that the user can issue to proceed the task.

Table 6.1: Task categories of the AIDL task tree notation.

6.2.1.2 Task properties

Beneath the task type, a task can be further specified by assigning several general properties to the task that are listed in table 6.2. Each of these properties can be specified for each task type: abstract, interaction in and out as well as application tasks.



Iteration	Iterated task. The task is continuously reactivated after it has been successfully performed. The iteration cycle can only be stopped by a disabling task.
Identifier	Each task has an unique identifier that should be specified in a human-readable format containing at least one verb describing the main action of the task.
Description	A detailed textual description of what the task is about.
Domain Objects	To perform a task, information might be required that is encapsulated into domain objects. Domain objects can be accessed, modified or created by a task. Only those domain objects should be specified (a) that contain the information that are required for successfully performing the task and written (b) that might be required by other tasks.
	<i>Connected tasks</i> allow to connect tasks of different trees together. A task connection can only be specified between tasks that describe the same problem to solve and therefore are named by the same identifier. The task type of each connected task can be considered as irrelevant for the connection since it only specifies who is in charge for the tasks: solely the application or the user by interacting with the system.
	<i>Asynchronous tasks</i> are running as long as they get explicitly disabled by another task. Until they are running they have access to the domain objects that are associated to the task.

Table 6.2: Task properties

Each task has to have an unique identifier that unambiguously identifies one way of performing a task, which usually consists of at least one verb that gives an impression about the main action that has to be executed to perform the task. A more comprehensive textual description can be added as another task property, which is most important for abstract tasks that have not been detailed any further. A task defines, reads, modifies or creates a set of domain objects containing the actual state of the task performance. The information is required while performing the task as well as to describe the result of the task's performance that might be relevant for subsequent tasks. The iteration task property allows to specify continuous reactivations of a single task respectively the contained task sequence of all of its children tasks. We do not specify the concrete number of iterations, because this depends on the user. To disable the iteration the user has to explicitly perform another task that is connected by a disabling operator to the iteration tasks on the same level of abstraction.

For modeling parallel flows of control we distinguish between connected and unconnected tasks. A task connection allows to connect all tasks with the same identifier to be executed or presented to the user just as one task. As soon as a connected task has been performed all connected tasks are marked as done as well. We use this way of synchronizing task performance within tasks of a single task tree to enable parallel lines of control, between two or more trees that run in parallel to synchronize simultaneously running applications, or to prototype new forms of interactions (see section 6.2.1.4). The task synchronization does not behave like the principle of a semaphore that waits for all other connected tasks with the same identifier. Instead it works related to the actual state of the application, which is basically compromised of all those tasks that are enabled at the same time. Therefore only in the case that two or more tasks with the same identifier are part of the same state, then all of those tasks that have the connection property set are combined into one task and performed together.

Usually the actions that a task comprises of are specified to happen synchronously, which means that a task's life cycle starts by collecting all the information required to perform, then the task does some internal calculations (application task) or requires the user to enter some information (input interaction task) and finally returns the results that might be relevant for the subsequent tasks or doesn't return any results if it gets interrupted (by a disabling task). Asynchronous tasks behave differently since they do not contain a defined point of return. There is no defined state for asynchronous tasks that can be used to identify the task as successfully performed. Asynchronous application tasks for instance can describe ongoing discovery mechanisms that are continuously looking for newly attached devices, or in case of an output interaction task continuously update their presentation to include new or updated information like for instance to display continuously updated charts. For input interaction tasks, the asynchronous property states that for example each keystroke that is used to enter information in a form field is directly send as an intermediate result to its associated domain objects and therefore can be accessed by any other asynchronous task.

Input Interaction Task's properties Input Interaction Tasks (IIT) can be further detailed by three different properties which are listed in table 6.3. The edit property details an IIT to be comprised of an editing action that enables the user to manipulate the associated domain object information. Control IIT are awaiting a command from the user to be successfully performed. Finally selection IITs require the user to select elements of a list to be performed successfully. In our understanding the selection contains a way to confirm the selection by the user at least when multiple selections can be stated and does not require an additional control task to confirm the selection.

Edit	Edit specifies that the tasks consists of a set of elements that the user has to manipulate, to enter and to confirm to accomplish the task.
Control	A control task awaits a command from the user to succeed.
Selection	A selection input interaction tasks requires the user to select elements from a list of elements. We distinguish between two different types of selection: single-choice or multi-choice. The former one requires the user to select one of a set of elements to perform the tasks, whereas the latter one requires selecting at least one element from a list and a confirmation of the user that the selection is finished.

Table 6.3: Task properties that can be associated to a task tree.

Supported domain object properties For each domain object that has been associated to a task the following properties that have been listed in table 6.4 can be specified:

<i>Identifier</i>	An unique identifier for an object, usually a noun, that gives the designer a clue about what is stored within the object.
<i>Concept</i>	For each object a concept can be defined and further detailed along with the abstractions offered by the task tree.
<i>Operation</i>	Four types of operations can be set for an object. Declare (D) defines the scope for a new object that can be created (C), modified (M) or read (R) by a task.
Description	A textual description of what the object is considered to contain and what it is used for.
Iteration	If a task is marked as iterative, one object, which contains a list or set of objects can be marked as iterative. In this case the tasks is iterated once for each object in the list.
Connected	The object is synchronized with all other tasks that are performing on the same object. If more than one tree is processed by the user the objects with the same identifier in both trees are synchronized as long as their tasks are enabled.
Query	A query is used to identify the relevant information of a domain object that are required for performing the task.

Table 6.4: Domain object properties that can be associated to a task tree.

For all objects at least the identifier property has to be set in order that it can be identified unambiguously. Further on, for each object a concept like a type or a class has to be set. Additional a textual description can be attached that describes what the

domain object is targeted to contain. Similar to connected tasks, the connect property of a domain object can be set to synchronizes it's contents to all objects with the same object identifier of the same or any other task tree that runs in parallel. In general all domain objects that are associated to asynchronous tasks are set to connected objects as default.

To detail domain objects of a task that is specified as an iterative task, objects can be set to be iterated together with the task's iteration. In this case the amount of iterations is bound to the number of elements in the domain object. Usually only one object is set to iteration but if more objects are set to be part of the iteration the number of iterations depends on the number of elements in the domain object.

6.2.1.3 **Annotating domain concepts and modeling the object flow**

Utilizing task trees as the workflow definition of a user interface at run-time also requires a stronger and more detailed connection of task and domain concepts. Such a connection on the one hand allows the identification of inaccurate domain abstractions not conforming to the task abstraction as well as the review of the distribution of concepts over the whole task tree and on the other hand provides the basis to derive meaningful user interfaces based on the task tree definition. While CTT already allows the annotation of objects and the marking of an LOTUS operator as 'information passing' [Mori et al., 2004], this approach is limited to the object synchronization between two tasks. Complex systems require modeling of the object flow throughout the complete task tree, which considers the object synchronization between all tasks that are enabled and are using the same domain concepts. Following the approach of [Klug and Kangasharju, 2005] we annotate the required objects as well as the access type directly to each task of the task tree allowing a fine grained modeling of the object flow.

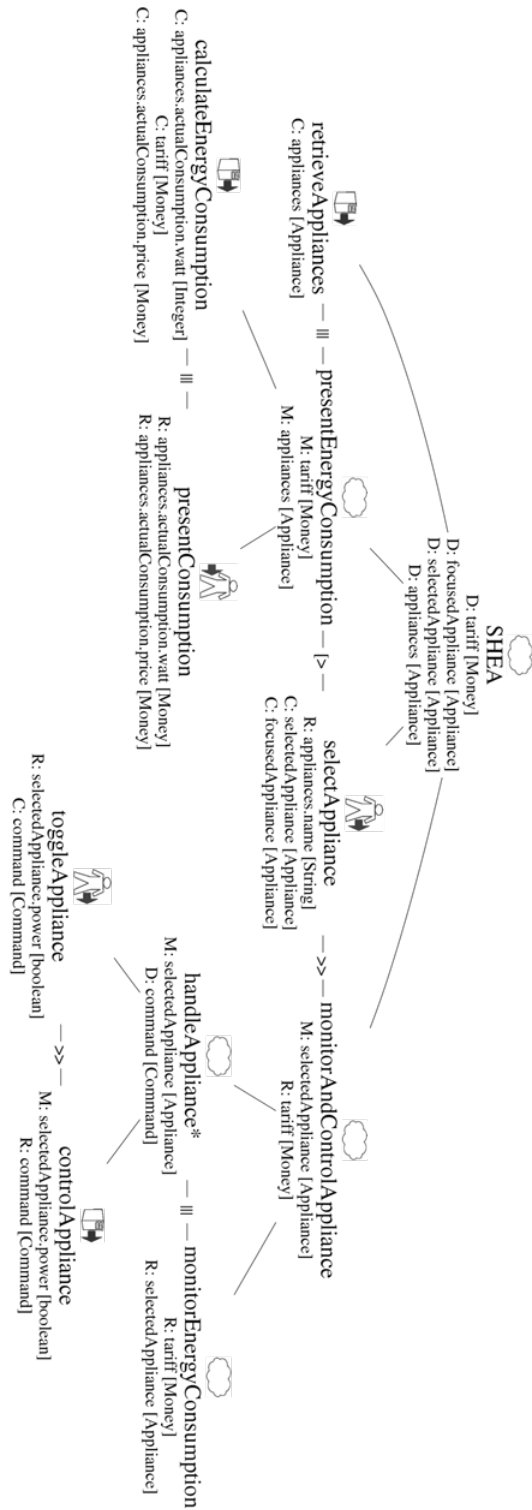


Figure 6.1: Abridged task tree of the SHEA application.

An abridged version of the task model of an application to support measuring the energy consumption (SHEA) is depicted in figure 6.1. The application discovers all appliances, which we have installed in an environment and presents an overview of all appliances to the user. By selecting an appliance the user can control and inspect past energy consumptions as well as extrapolations of future consumptions by issuing voice commands or using a wall-mounted touch display. Like depicted in figure 6.1, the access type is limited to read (R), modify (M) or create (C) operations. By specifying the declaration (D) of each object before the first usage we support the definition of a scope for each object. Similar to a programming language, each object has to be declared before it can be created and created before it can be modified or read. Under each task's name, we annotate the domain object type in square brackets (by referencing a class name) together with the domain object name, whose attributes can be accessed by using a dot followed by the attribute name. During interpretation of the task tree at run-time the domain object annotations are used to synchronize the enabled tasks objects, which are actually performing. Thus, if one task modifies a domain object and another task that is enabled concurrently has read access to the same domain object, the task receives the object's modifications instantly. Further on, the declaration of objects enables the run-time environment to keep track of objects and to remove objects that are no longer required as the user continues the interaction. Inaccurate domain abstractions that do not conform to the task abstraction can be identified easily by checking the cluttering and granularity of the object annotations. Furthermore basic automatic consistency checks (i.e. create before modify and read) across multiple subtasks are possible, because declarations as well as all defined operations are also summarized by the next higher level of abstraction in the task tree. The designer can further partition and detail domain concepts in sub-trees together with the tasks performing on the concepts. This ensures that the resulting interactive application is not switching between several concepts too often, which make the application complicated to use, since it raises the cognitive workload of the user. If the same concepts are spread over the whole tree re-modeling the task tree should be considered, regarding two aspects. On the one hand the conceptual abstractions should be reviewed and it should be checked, whether these concepts can be detailed conforming to the associated task abstractions. On the other hand the domain object distribution should be solved by moving tasks that are performing on the same domain objects together into the same sub-tree.

6.2.1.4 Interconnecting Task Trees at run-time

To support the flexible addition of new modalities at run-time, we allow tree concatenation by using abstract tasks as leaf nodes of a task tree. During interpretation of the tree, the referenced node is added at the location of the original abstract task.

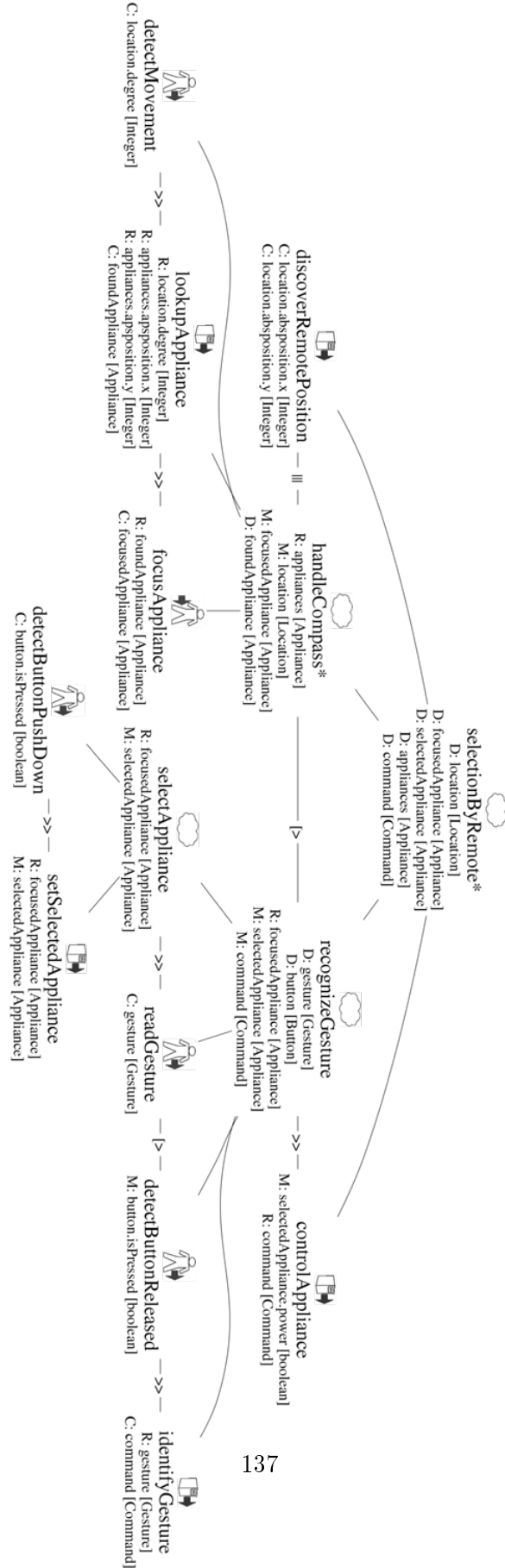


Figure 6.2: A modality specific task tree that describes a remote control.

Figure 6.2 depicts a modality specific task tree we have prototyped for the SHEA application that describes a remote control that is aware of the direction it points to and is able to detect gestures that are interpreted as appliance control commands. The remote requires the user to point to an appliance first, and then waits for the user to press the remote's button, which starts the gesture detection and initiates a command for the selected appliance that is issued after the button has been released by the user.

The modality-specific tree can be interconnected by the designer with the application tree using two synchronization methods: On the one hand domain objects of the application task tree can be synchronized with the modality specific task tree. This is specified by annotating the same domain objects names to the modality specific task tree. For the task tree describing the remote control the object that the remote is pointing to (see figure 6.2: `focusedAppliance`) gets synchronized with a presentation of the focus in the SHEA task tree (the `selectAppliance` task of figure 6.1). Further on, all discovered appliances (stored by the `appliances` domain object) as well as the actual selected appliance (`selectedAppliance` object) get synchronized between the remote and the SHEA application.

On the other hand tasks of the modality-specific tree can be connected to tasks of the application task tree. In our example we use task synchronization when the user has to choose the application she wants to control (in the `selectAppliance` task). Since both trees are started simultaneously the first task that the user is required to perform in the SHEA application is to select an appliance to monitor its energy consumption history. Following the design of the application task tree this has to be done by choosing one appliance of a list of appliances. Using the remote the user just needs to point to the appliance (`handleCompass`) and press the remote's button to select it to perform the `selectAppliance` task. We interconnect both `selectAppliance` tasks using a directional connection, which is only enabled if the connected application task tree's task is enabled and does not require changes to the application task tree. The remote can be used independently but if both trees are in the same state, requiring the user to select an appliance both `selectAppliance` tasks can be either performed by using the remote or the touch display originally supported by the application task tree.

In our current implementation we have implemented support for interconnecting IIT and support every temporal operator of the CTT notation for interconnected tasks. Interconnecting application tasks is supported as well (see `controlAppliance` task in both figures) but only describes that a call to the functional core has to be done once only. The modality-specific tree should follow the basic interaction flow of the application task tree since this eases the complete integration of the prototyped interaction with the new modality into the application later on. Following a prototyping approach, not the whole application is required to be supported by the modality-specific task tree. Thus, in the beginning only parts can be covered by the new modality and can be continuously evolved during the prototyping process.

The order of all interconnected tasks must remain the same for both, the application task tree and the modality-specific task tree. In our actual implementation the remote of figure 6.2 cannot be designed to support initiating the control command before selecting an appliance, since this will break the design of the SHEA application. But requiring

the same fixed sequence of interconnected tasks for all modality-specific trees has the benefit, that the modality specific task tree can be merged with the application task tree after the prototyping process has been finished.

6.2.2 Abstract User Interface Model

A domain model is often mentioned as a basic model for a model-based approach [Limbouurg and Vanderdonckt, 2004a, Puerta, 1997, Calvary et al., 2003] on the conceptual level. Since the domain model is developed by software engineers and given “as-is” [Limbouurg, 2004], most model-based user interface development approaches are not focusing on the characteristics of a domain model explicitly but concentrate on the mapping of a domain model to the other models like for instance done by [Pribeanu, 2002]. Following such an approach requires to define a concrete conceptual model for the domain model for which the mappings can be explicitly defined, such like UML diagrams or Extended Entity Relationship Models [Teorey et al., 1986] as done for instance in UsiXML [Limbouurg et al., 2004b].

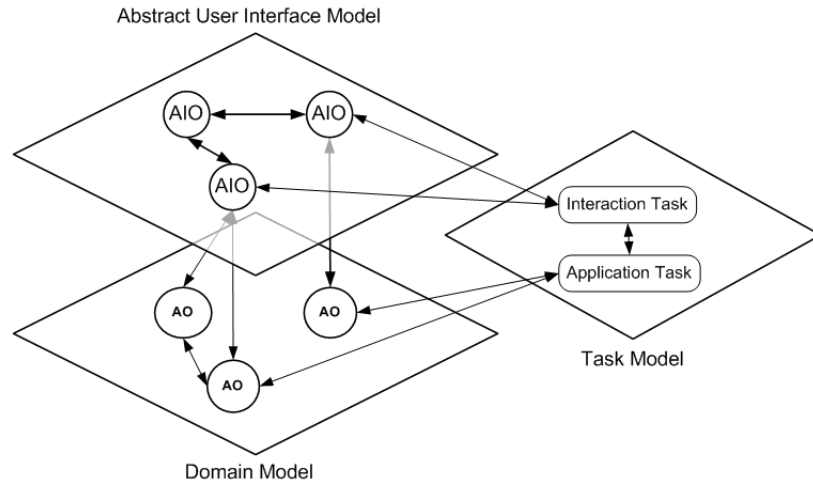


Figure 6.3: The abstract user interface model is realized as a filtering view over the domain model. In the task model application tasks refer to application objects (AOs) of the domain model and interaction tasks link to abstract interaction objects (AIOs).

We follow a different approach that does not define mappings between concrete attributes of the conceptual domain model, but between application objects (AOs) of a domain models like illustrated in figure 6.3. We understand an AO as a discrete object that can be of any complexity and can be associated via various relationships to other AOs. Different technologies and languages can be utilized for the realization of the domain model. Only application tasks of the task model can be linked to the entities of the domain model, whereas interaction tasks can only be connected to abstract interaction

objects (AIOs). AIOs map one or more AOs similar to a filtering view, which is gained by applying the selection activity (and are enriched later on).

We use ontologies to describe the domain model mainly because of three reasons:

1. An ontology can be understood as a general description of concepts and relationships of a data model and are well suited as a basis for knowledge representation as they usually are able to describe individuals, classes attributes and relations in *one* language.
2. Common ontology languages are declarative and every concept or relationship is explicitly stated. This makes an implementation of a domain model that is based on ontologies easy to realize, as there is no effort required to extract the concepts and relationships from a source code, which might be required if we choose a programming language.
3. Most ontology languages are very expressive compared to other data models like database structures or static models described by object-oriented programming languages, since they do not limit or restrict for instance relationships like done in some programming languages for instance regarding support for multiple-inheritance relationships.

Beneath using ontologies other technologies can be used for a domain model. For instance in the cooking assistant case-study that we present in chapter 7, we use Universal Plug and Play profiles as part of the domain model for controlling home appliances.

As illustrated by figure 4.10 depicting the abstract user interface derivation process, the first step to derive the abstract interaction objects from the application objects is to apply the *selection activity* to mark relevant classes and attributes of the application objects. For doing this, we use a filter definition that can be applied similar to a template mask, separating the data structures relevant for interaction from the rest of the data structures of the domain model.

Structure	Application Object	Basic Interaction Object
Internal	Inheritance	Attribute with namespace Frame with namespace List with namespace
	Composition	Frame, list Attribute
external	Association	List
	Aggregation	Frame, list

Table 6.5: The table lists mappings between concepts of application objects and interaction objects.

We distinguish between two types of structural mappings. External mappings consider the relationships between objects, whereas internal mappings focus on the mapping of

concepts that are internal to an object. As we realize the selection activity by applying statements describing how to extract the relevant data from the application objects, the basic statements for interaction object derivation are listed in table 6.5. We propose three general types of selection statements: The attribute statement selects a primitive of an entity, the frame statement is used to select a complex entity containing internal relations and attributes, and the list statement is used to select sets and lists of primitive or complex entities.

Depending on the types of relations and what types of entities are related, we choose the selection statements:

Composition describes a containment relationship, where an application object contains one or more other application objects. For the composition of primitive elements, the attribute statement, for composed collections the list statement, and for complex elements the frame statement is used.

Inheritance describes a relationship where one or more application objects are detailed and refined by another application object type. Thus, the inherited object contains the information of its parental application objects as well as additional ones. The inherited information can be selected by adding a namespace definition to the selection statement. A namespace can be attached to an attribute to refer to a primitive of a parental entity.

Aggregation describes a weak part-of relationship between entities. The entity that aggregates the other entities usually contains more complex functionalities that it delegates to the aggregated entities. Thus the aggregation has to be mapped using an external mapping, since the aggregating entity does not have direct access to the aggregated entities but delegates operations.

Association Different to UML that defines an composition and aggregation as specialized variants of an association, we address the association separately by considering all associations between entities that are not describing a composition or an aggregation relationship. In this reduced category of associations all structural mappings to be defined are external, which means that they are specified about a relationship that relates two or more entities without having explicit access to the related entities internals.

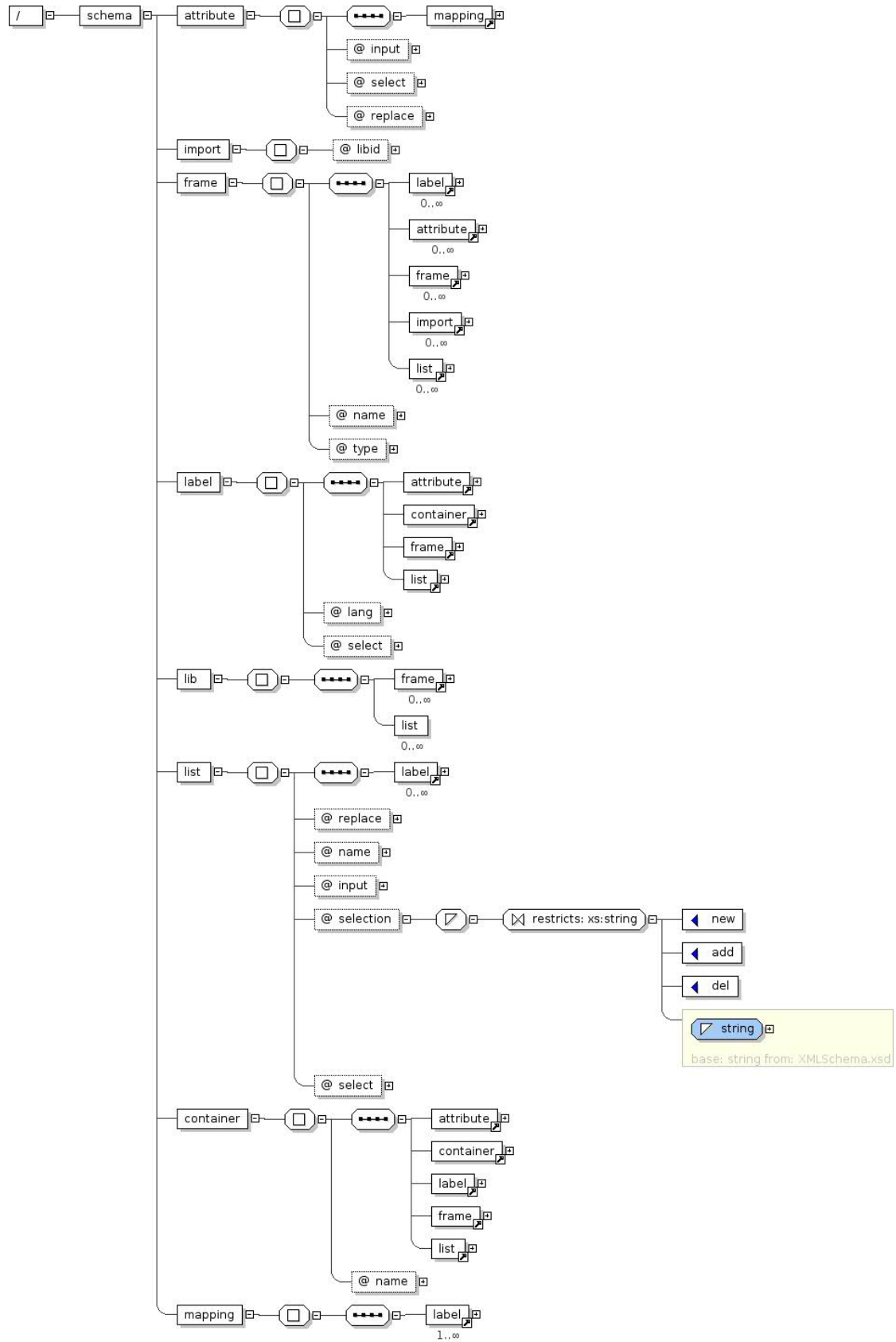


Figure 6.4: The XML Schema definition of the abstract user interface model.

Figure 6.4 presents a compressed view of all relevant elements and attributes of our abstract user interface model.

In the following, a small example is discussed to demonstrate the derivation process and to introduce our notation to select relevant AO information and enrich the basic AIOs with further information.

6.2.2.1 Selection of Basic AIOs

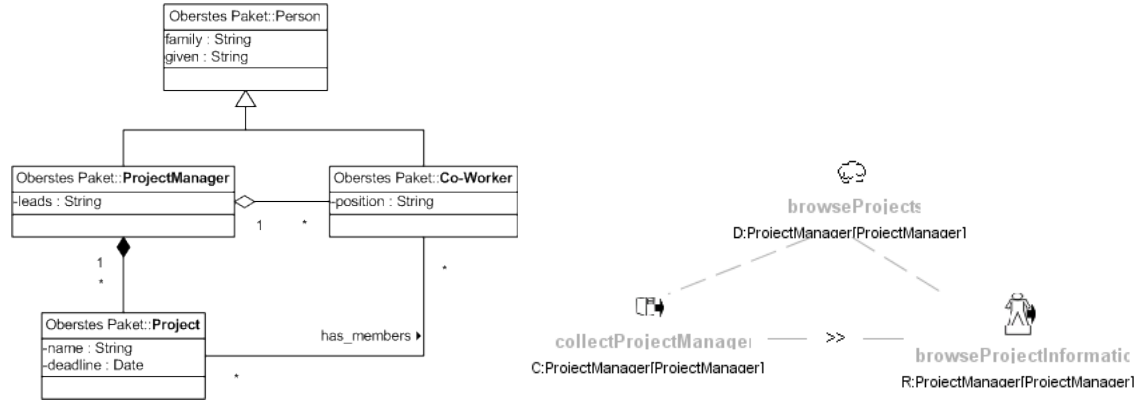


Figure 6.5: UML class diagram of an example application object structure (left). The basic task tree for the project browser interactive system that consists of an application task that sends application objects to an interactive task (right).

Figure 6.5 depicts an exemplary application object structure that should be mapped to a set of interaction objects. For this example we only consider a very basic task tree, describing an interactive system that enables to browse all project managers, their co-workers with names and positions and the projects the project manager has started so far. During the analysis phase we have identified the application objects Project Manager, Co-Worker and Project as relevant for the presentation in the user interface. After that we found an already existing service in the task modeling step of the design phase that is able to provide the required application objects in form of a data structure like depicted in figure 6.5. We decide to reuse the service and integrate it in our interactive system using an application task. The downside of reusing the service is that it is implemented to return just a set of Project Managers. Every information about the Projects of the ProjectManager has been considered as a composition that is private to the Project Manager and the relationship to his Co-Workers has been implemented using a reference. Therefore we specify the collectProjectManagers tasks to produce one output application object “ProjectManager”, which is a set of objects from the type *ProjectManager*. The interaction task *browseProjectInformation* is set to receive the set of *ProjectManager* as an input application object.

Now the ProjectManager objects application objects are used to derive suitable interaction objects for the user interface of the interactive task. We start by selecting the

relevant attributes that are required to be presented in the user interface. The selection process works by constructing a filter that hides every information not relevant from the original application object but preseveres the original structural relationships.

Listing 4 Selecting objects using our abstract interaction language.

```

1 <list name="project_manager_ao">
2 <frame type="Project:DAI_1.ProjectManager" name="project_manager_ao">
3 <attribute select="leads"/>
4 <attribute select="Project:DAI_1.Person.family"/>
5 <attribute select="Project:DAI_1.Person.given"/>
6 <list select="projects">
7   <frame type="Project:DAI_1.Project=" name="projects">
8     <attribute select="name"></frame>
9 </list>
10 <list select="coworkers">
11   <frame type="Projects:DAI_1.Co-Worker" name="coworkers">
12     <attribute select="family"/>
13     <attribute select="position"/>
14   </frame>
15 </list>
16 </frame></list>

```

Listing 4 shows an exemplary definition of a selection filter suitable for marking the relevant attributes that should be considered for the presentation of the user interface for the *BrowseProjectInformation* interaction task. We use the *frame* language element to select the objects associated by aggregation and composition relationships. The “attribute” element is used to select attributes that are contained in an application object. The *list* element is able to iterate through attributes or objects consisting of a set or a list structure. Like shown in the exemplary source code fragment of the filter definition the original basic structure of the abstract interaction object is preserved since attributes and lists can only be selected in their corresponding frames and the frame nesting reflects the original aggregation and composition relationships of the original application object.

6.2.2.2 Enrichment of Basic AIOs

After all attributes have been selected, the filter definition can be re-structured within the limits defined by the frame statements for instance by changing the attribute order. Additional containers can be used in order to sub-group complex frame statements or merge various frames that are specified on the same level. Listing 5 presents the final abstract user interface source code after the restructuring and enrichment steps.

Listing 5 Structured and enriched abstract user interface.

```

1 <list name="project_manager_ao">
2   <frame type="Project:DAI_1.ProjectManager" name="project_manager_ao">
3     <container name="manager">
4       <label lang="en">Project Manager:
5         <attribute select="Project:DAI_1.Person.given" replace="true"/>
6         <attribute select="Project:DAI_1.Person.family" replace="true"/>
7       </label>
8     </container>
9     <container name="projects">
10      <label lang="en">Manages the following projects:
11        <attribute select="leads"/>
12        <list select="projects" replace="true">
13          <frame type="Project:DAI_1.Project=" name="projects">
14            <attribute select="name" replace="true"></frame>
15          </list>
16        </label>
17      </container>
18      <container name="team">
19        <label lang="en">The team of the Project Manager:
20          <list select="coworkers" replace="true">
21            <frame type="Projects:DAI_1.Co-Worker" name="coworkers">
22              <label>Name:<attribute select="family" replace="true"/></label>
23              <label>Position:<attribute select="position" replace="true"/></label>
24            </frame>
25          </list>
26        </label>
27      </container>
28 </frame></list>

```

Beneath attribute re-ordering, two new languages elements have been utilized. The filtered attributes have been structured in greater detail using a *container* statement. Using this statement, the data has been segmented into three groups: “manager”, “projects” and “team”. Each group combines all attribute and frames that should be considered as strongly connected together and should not be presented separately to the user. Different to container statement the *label* element groups and links contextual information to attributes, frames or a set of frames and attributes in order to support the user in understanding the presented information. Everytime attributes or frames will be presented to the user, all enclosing contextual information that are expressed by label statements are considered.

6.2.3 Layout Model

A basic source of information for layout constraint generation is the task model since its structure allows deriving containers consisting of tasks that can be performed in parallel - a presentation task set (PTS), which is derived based on temporal relationships of the

tasks that are simultaneously enabled.

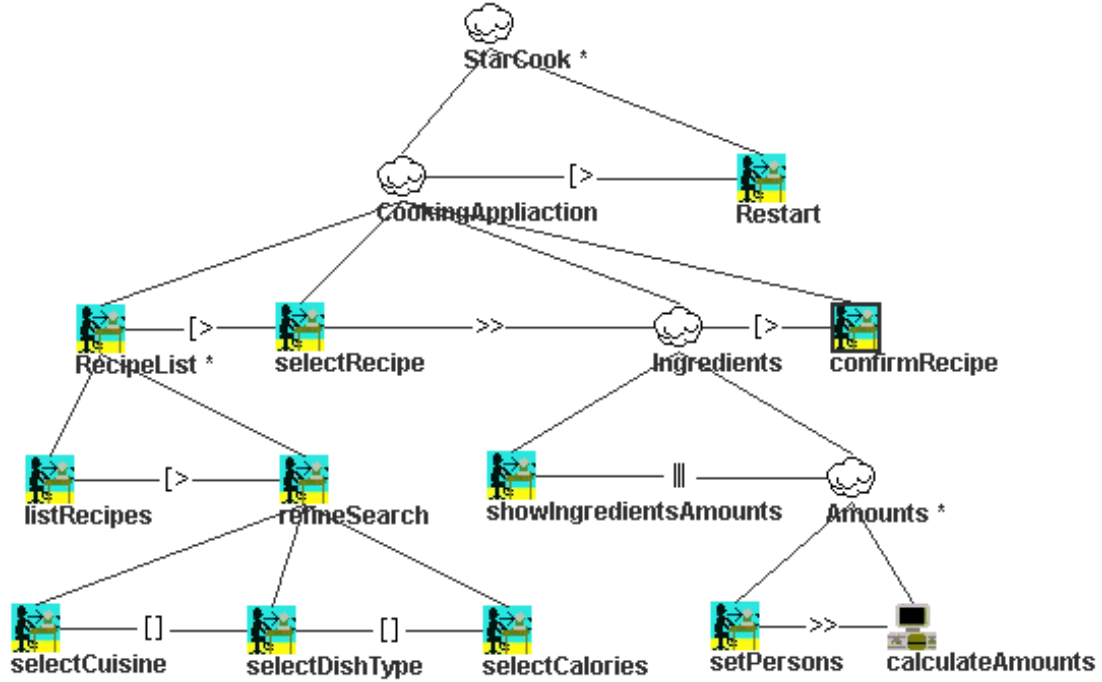


Figure 6.6: The recipe finder part of our cooking assistant.

Figure 6.6 illustrates one part of a cooking assistant, the recipe finder, which describes how the user can search for recipes and calculate the required amounts of ingredients for a given number of persons afterwards. Definition (1) specifies one possible PTS of the cooking assistant task tree:

- (1) $\text{PTSRecipeFinder} = (\text{listRecipes}, \text{selectCuisine}, \text{selectDishType}, \text{selectCalories}, \text{selectRecipe}, \text{Restart})$

We use the task hierarchy to derive the container's nesting structure by iterating through the levels of the tree and generating constraints, which specify a container for each non-atomic task to contain all its children by the function:

- (2) $\text{Containment} = \text{iterateTaskTree}(\text{TaskModelCook}, \text{PTSRecipeFinder})$

A function to generate an initial sizing of the containers and their corresponding elements is:

- (3) $\text{Size} = \text{weightByAtomicTaskNodes}(\text{TaskModelCook}, \text{PTSRecipeFinder}, \text{priority});$

This function iterates along the abstraction levels of the task tree, counts for each abstraction level the amount of atomic tasks that are part of the PTS and divides this amount from the total amount of tasks in the PTS. This way it calculates the weight for each container and element. Like illustrated in figure 6.6, *CookingApplication* task contains five atomic tasks that are in the PTS, of which four are children of *RecipeList* and the remaining one is *selectRecipe*. Thus the container size is calculated as $4/5$ for *RecipeList* and $1/5$ for the *selectRecipe* task. Whereas functions (2) and (3) describe exemplary general model-based layout derivations, we generate constraints for the orientation of the elements based on context-of-use by:

(4) Orientation = alternateOrientation (TaskModelCook, PT-SRecipeFinder, Platform Model:AspectRatio)

Function (4) reads the actual PTS from the task model and switches the orientation of the elements by toggling horizontal to vertical and vice-versa, for each abstraction level of the task tree. While changing the orientation for each container produces elements with a balanced width to height relation, function (4) additionally considers the platform model and sets as much tasks horizontal or vertical during alternating the orientation to optimize the layout for a specific aspect ratio at run-time. As for the recipe finder an optimization for vertical orientation (to optimize for a 4:3 screen aspect ratio) results in layouting *CookingApplication* over *Restart* (vertically oriented) and *RecipeList* left of *selectRecipe* within the *CookingApplication* container. While functions like (2)-(4) can be used to generate an initial layout setup that is consistent to the design models or can be used as fallback to layout user interface distributions that have not been specifically addressed during design-time, a model-based layout generation can only offer an interpretation of these models. Thus, we implemented a layouting model generator that a designer can use on the one hand to simulate the final user interface layout and possible context-of-use scenarios and on the other hand to select and manipulate suitable constraint generation functions that should be part of the application layout model.

7 The Cooking Assistant Case Study

The design of the method, the MASP architecture, and the language additions that we have presented in the last chapters have been conducted as part of several research projects. All research projects involved industrial partners and have been partly or completely sponsored by industry and the German government. We started the initial research during the design of the JIAC IV software agent environment. JIAC IV includes a method to design software agent systems but missed a sound approach to design interactive applications. We presented the initial concept for enabling access to agent-based services in [Boese and Feuerstack, 2003] and implemented a first prototype that was named the Multi-Access Point as part of the JIAC IV project, which was sponsored by the Deutsche Telekom. Based on the initial prototype two research projects, the BerlinTainment [Rieger et al., 2005] project, which was sponsored by the German federal ministry of research and education and the Personal Information Assistant that was initially sponsored by the Deutsche Telekom utilized the MASP to implement graphical user interfaces. In both projects services were offered to be run on a desktop PC as well as on several mobile devices. Further on, we conducted early experiments to generate speech-based interfaces, which have been developed as part of the BerlinTainment and its followup project, the Smart Entertainment Assistant. Most of the feedback we received from these projects was about the complexity of developing multi-platform user interfaces, which forced us to further develop the method to systematically design and implement user interfaces based on a model-driven approach.

During the Seamless Home Services project [Feuerstack et al., 2006], sponsored by the Deutsche Telekom, we created several interactive applications that can all be accessed via voice and graphical user interfaces. By following the MASP method a personal video recorder (PVR), a personal newsletter service, and a home automation system have been successfully realized. As part of the Seamless Home Project we extended the MASP Builder tool and improved the models of the MASP to better support the generation of multi-platform user interfaces.

As the space is limited for describing all these projects, we will instead describe in the following sections one complete application of the MASP methodology in greater detail. We apply our method to develop a cooking assistant service, an application that has been developed as part of the Service-Centric Home (SerCHo) project. SerCHo is a research project that has been sponsored by the German federal ministry of economics and technology. The SerCHo consortium involved seven partners from the industry working together with the Technische Universität Berlin.

The initial idea of the cooking assistant service is to motivate consumers watching cooking shows on television to learn and improve their cooking skills based on following the recipes and advices they have watched in the cooking show. As the cooking assistant

is targeted to support the users during cooking, it was important that it can be controlled intuitively using voice and several graphical displays.

In the following sections we will run through all the steps of the MASP methodology that have been described in chapter 4. Our method starts with writing textual scenarios compromising the initial input for the method (section 7.1). Thereafter, we are extracting relevant tasks and concepts from the scenarios in the analysis phase in section 7.2. In the next section 7.3 we describe the design phase by modeling the designated task flow of the cooking assistant with the help of the MASP Task Tree editor and simulate the cooking assistant usage through our Task Simulator to test if the modeled task flow meets the user demands and check if all scenarios have been sufficiently considered. In section 7.4 we model the domain objects that have been identified during the analysis phase using the JIAC ontology editor and generate some exemplary objects like for instance recipes to check if all required data can be captured by the domain model using the fact editor of Protégé. Domain objects are derived into abstract interaction objects to be represented or manipulated by the user interface in the abstract user interface design in section 7.5. A tool, the MASP Builder supports the designer in selecting the relevant objects as well as arranging and enriching the abstract interaction objects. Section 7.6 describes how we derive the layout model for the cooking assistant based on the other design models to adapt the assistant to several different display sizes allowing to access it from any computer and touch display in the apartment. In section 7.7 we describe the derivation process of user interfaces for various platforms like web browsers or voice browsers by using a multi-level transformation process to derive the concrete user interface. Finally, we describe how we can connect the user interface to the environment, using the service model in section 7.8, deploy the models into Model-Agents in section 7.9 and conclude by summarizing the results and lessons that we learned by the case study in section 7.10. During the description of the development process of the cooking assistant we refer to screenshots and small source code excerpts to explain the details. Further information is collected in the annex chapter where we collected all the screenshots of the main screens of the cooking assistant. Further on the complete task model is attached to get an impression about the whole cooking assistant application.

7.1 Analysis Phase: Writing scenarios

We start the development process by collecting and writing scenarios describing the basic features of our cooking assistant. The cooking assistant includes a recipe finder to search for recipes based on various querying options. Further on, a cooking aid that guides the user through the cooking steps, advises him about the required amounts of ingredients, gives context sensitive cooking hints in form of video presentations, and allows controlling the kitchen appliances to ease the cooking process.

The cooking scenario

7 The Cooking Assistant Case Study

Actor:	Ann
Start state:	Ann in living room.
Defining goal:	Cook a recipe that has been watched in a television show.

Ann likes to watch her favorite cooking show in television and often thought about preparing one of the meals on her own. Until now she always is discouraged by the complexity of the recipes. Further on it often happens that Ann forgets during shopping about some of the recipe ingredients and the required amounts. But now she has installed the Multi Access Service Platform, which she has connected to her home infrastructure and her television. On the platform she has installed the cooking assistant that is accessible via a wall-mounted touch screen in her kitchen, on a television, her cell phone and can be controlled via voice as well.

When Ann presses the “red button” of her remote, she can directly switch to her cooking assistant and gets presented an overview allowing her to choose between cooking support for the recipe that she has just seen on *television*, a *daily recommended healthy meal* or to *search for a recipe* on her own.

Ann selects to search for a recipe on her own. She can browse through her personal recipe database and *filter the recipes* by three main criteria: First, the dish type supporting to choose between *main dish*, *dessert*, *starter*, and a *snack*. Second, she can select between four different nationalities: *German*, *Indian*, *Chinese* and *Italian* food and finally, she can search for *diet*, *healthy* and *normal meals*. Ann searches for a normal German main dish and decides for “lamp chops”, which she likes most from the resulting list of recommendations.

After Ann has decided for the “lamp chops” to cook, her cooking assistant asks her for the number of persons she wants to cook for and calculates the required amounts of ingredients. Ann selects to cook for three persons and answers the question of the cooking assistant if she wants to generate a shopping list with “yes”.

Now the cooking assistant goes through the lists of required ingredients and asks Ann for each ingredient if she has enough of it available in her kitchen. Ann is not sure if she has everything available, so she leaves the living room and goes into the kitchen. The cooking assistant notices the room change of Ann and adapts the user interface to the kitchen’s touch screen.

Ann goes around in the kitchen and looks for each ingredient. For ingredients that she has available she just answers “is available” whereas for ingredients Ann needs to buy, she just answers “not available” or “only half a liter available”.

From Ann’s answers that she can give via voice prompts or by using the touch screen, the cooking assistant generates a shopping list and offers Ann to send the list to her cell phone. On Ann’s command the cooking assistant distributes the shopping list to her cell phone, on which the migrated user interface part still remains interactive and allows Ann to mark each ingredient as available as soon as she bought it at the local mall.

Ann has only a few experiences in cooking meals but she owns a well equipped kitchen, which includes an *oven*, a *chimney* and a *stove*. Every kitchen appliance can be *controlled* via IP, but she often fails to get advantage of this equipment as she doesn’t like spending her time reading huge and complex manuals just before preparing a meal.

After Ann arrives back from shopping she continues to use the cooking assistant that now presents the expected preparation time and the cooking advices in a step-by-step format. Each step includes a detailed description, links to the relevant part of the cooking show video and enables a one-click control and the automatic programming of the relevant cooking appliances. For her convenience the cooking assistant further *highlights* the *ingredients* to ensure that she does not miss one. If Ann is unsure if she has finished all required actions to succeed a step, she can ask for the ingredients by speaking “list ingredients”, the required ingredients amounts by “amount of lamp chops”, ask the cook to re-read the detailed step description again by “read description”, or let the cook play the relevant part of the cooking show by asking the cooking assistant for help by “play video”.

7.2 Analysis Phase: Identifying tasks and concepts

For each scenario we used the informal-to-formal task analysis approach to identify the relevant tasks and objects. For the main scenario that has been described in the last section we identified the following user tasks and objects that are listed in listing 6 and have been highlighted in the scenario description.

Listing 6 Identified tasks for the cooking assistant.

Watching television show
Shopping recipe amounts of ingredients
Controlling kitchen appliances (oven, stove and chimney)
Browsing recipes
Searching for nationality, dish type, and amount of calories
Writing down required ingredients
Recalculating amounts of ingredients for number of persons
Checking for missing ingredients
Striking out available ingredients
Cooking step-by-step
Highlighting ingredient amounts for each step

Using the Eliciting tasks tool [Paternò and Mancini, 1999] we can now identify the cooking assistance process structure by ordering the tasks by the time they usually occur and grouping tasks that belong together or can be done alternatively or in parallel:

Listing 7 Grouped Tasks.*Television*

Watching television show

Recipe Finder

Searching for recipes with the following criterias: nationality, dish type, and amount of calories

Browsing recipes

Shopping List

Writing down required ingredients

Recalculating amounts of ingredients for number of persons

Checking for missing ingredients

Striking out available ingredients

Shopping recipe amounts of ingredients

Cooking Aid

Cooking step-by-step

Controlling kitchen appliances (oven, stove and chimney)

Highlighting ingredient amounts for each step

Further on, we collect all relevant domain objects:

Object	Property
Recipe	Ingredients and amounts
	Nationality
	Dish Type
	Calories
	Number of persons
Steps	Required ingredients + amounts
Recipe Database	Recipes
Search criteria	Nationalities
	Dish types
	Number of calories
Appliances	Oven
	Chimney
	Stove
Shopping list	Required ingredients and amounts
	Available ingredients and amounts

Table 7.1: Relevant Domain Objects.

Now we combine the tasks (each tasks has to have a unique name) with the identified objects (each object has to have a unique name). All tasks must also include at least one object to perform and one object can be utilized by several tasks.

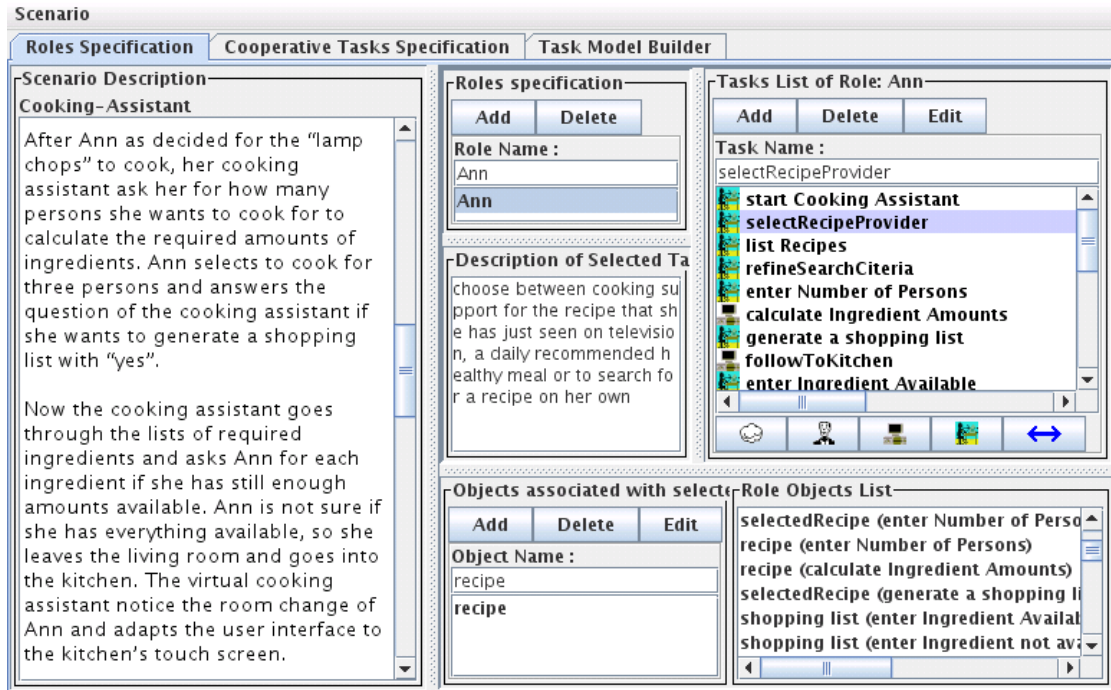


Figure 7.1: Screenshot of the eliciting task tool during the analysis phase.

Figure 7.1 presents a screenshot of the Eliciting tasks tool [Paternò and Mancini, 1999] that has been used to identify the basic tasks and objects during the task analysis phase. The tool requires loading textual scenario descriptions, which it shows on the left side of the tool in figure 7.1. Tasks, roles, and objects can be extracted from the scenario descriptions by marking the relevant words of the textual description and pressing the proper “add” button to store the marked text in a list of roles, tasks or objects. Tasks, objects and roles can be renamed to reflect shorter names. To link a task to a role or an object to a task, the corresponding role or task has to be selected before the marked text can be added. In figure 7.1 such a link is depicted where the role “Ann” has to perform the “selectRecipeProvider” task, which requires the “recipe” object to perform successfully.

After all relevant tasks, objects and roles have been identified all scenarios are re-checked during the testing step of the tasks analysis. Therefore each scenario is read again and the tasks list is checked for completeness. Typically those objects that have been identified and linked to tasks in the end of the first iteration need a second iteration to be considered for the tasks that already have been identified in the beginning of the analysis as well.

The analysis phase ends up with a set of roles, each performing a set of tasks where each task requires to read or to manipulate one or several objects. With this information we switch to the design phase. Different to the results of the analysis phase, the results of each step in the design phase can be directly interpreted in our run-time environment

to be tested, and forms one aspect of the final interactive system that is realized by a certain Model-Agent.

7.3 Design Phase: Modeling the task tree

We continue modeling the task tree by using our MASP Task Tree Editor (MTTE). Figure 7.2 shows a screenshot of the MTTE tool, which presents parts of the cook application’s task model. The MTTE supports all three basic steps of the task design phase: (1) task and object refinement, (2) identification of temporal relationships, and (3) object refinement and specification.

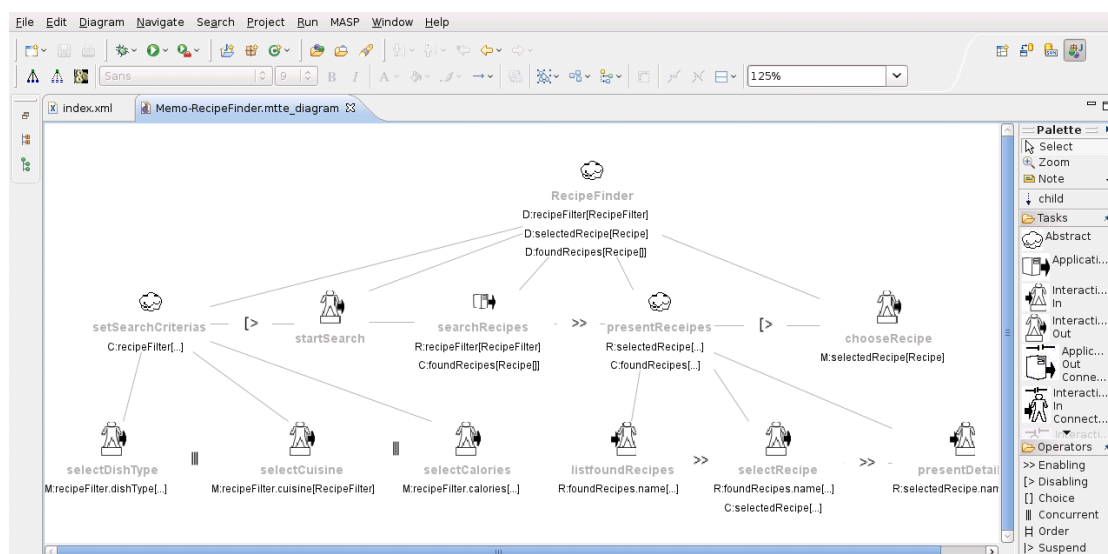


Figure 7.2: The MASP Task Tree Editor during the design of the recipe search process.

For task and object refinement (1), the MTTE is able to load the results of the analysis phase, which includes all tasks and their corresponding objects that have been identified so far. The MTTE presents all tasks using symbols specifying the tasks' types and annotates the corresponding objects' names directly under the task's name. By following the task design using the MTTE tool all tasks have to form a strict hierarchy, which requires modeling parent tasks describing more abstract tasks as well as detailing tasks that are too abstract to be directly executed within the application. In figure 7.2 such a task hierarchy is depicted: cloud symbols are used to specify abstract tasks that require a more detailed specification on a lower level of abstraction. This can be modeled by using the toolbar that is presented at the right of figure 7.2, allowing the designer to model abstract, application, interaction in, and interaction out tasks.

Temporal relationships (2) need to be defined for all tasks that share the same level of abstraction as well as the same parent task. Like depicted in figure 7.2 the MTTE supports modeling the “enabling”, “disabling”, “concurrent”, “choice”, “order independence”,

and “suspend” relationships. By using the temporal relationships, basic task sequences are defined that describe how the overall goal can be reached. Thus, for the RecipeFinder part of the task tree depicted in figure 7.2, the task sequence requires the user to enter the search criteria first and then allows him to initiate the search, which results in a list of recipes that the cook presents to the user. Finally the RecipeFinder part of the task model is successfully performed as soon as the user decides for one recipe to cook.

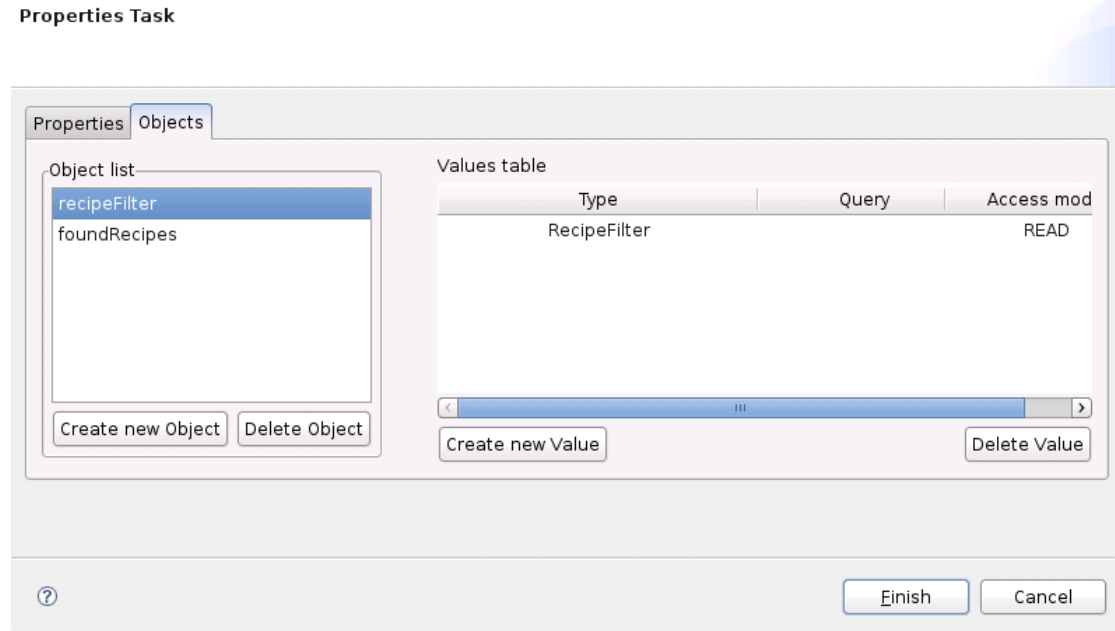


Figure 7.3: The screen for the object refinement in the MASP Task Tree Editor.

The objects that are required for performing the tasks (3) are refined corresponding to the task refinement. This is done on the one hand by following a top-bottom approach and by continuously detailing the objects and their attributes. For instance the “recipeFilter” object is defined locally for the “setSearchCriteria” task.

On the other hand by traversing bottom-up we ensure that all objects that are required by two tasks are referred by all parental tasks until a task abstraction level is reached where both tasks referring the same objects share the same parental tasks relationship. This top-most task then defines the scope of the object. For example in the RecipeFinder task tree the “foundRecipes” object is referred by several tasks requiring a global definition of the task object on the highest level of abstraction – the “RecipeFinder” task. During the object refinement all referred objects require an operation type, which can be set to “read”, “modify”, “create”, and “define”, as well as an object type, which is annotated for each object by using the square brackets.

Testing the task model

During the design of the task model, automatic model checking and interactive task simulation can be done to improve and evaluate the task model status. The former one is applied to reveal inconsistencies of the annotated domain objects' access operations and helps to identify re-usable domain objects. The latter allows simulating all of the task flows that have been specified so far and enables to check whether all task processes are designed consistent to the textual scenario descriptions.

For the cooking assistant case study we used the task simulator to present and discuss how the cooking process should be supported. By simulating several different possibilities we ended up separating the complete recipe into a sequence of steps, which was the most comfortable process for the users. Further on, it figured out that the video-based help, as well as the appliance control functionality should be offered context-sensitive to the actually active recipe step. This context-sensitivity has several benefits: The complexity of the user interface could be reduced as only those appliances and help is offered that is relevant for a certain preparation step. Additionally the appliance control could be reduced to a one-click action, as each step already includes all relevant information (for instance the heating temperature of the oven) to directly control each appliance on behalf of the user.

During the cooking assistant case study we reduced the amount of enabled tasks for certain scenarios by using the abstract user interface preview that is generated by the task simulator. For instance, the video-based help functionality required a lot of the screen space to be useful even if a person is not directly in front of the display. Thus we simulated several scenarios to identify those tasks, which can be suspended during the video-based help is accessed by the user.

Whereas the cooking assistant task tree is a rather straight-forward example (please refer to the complete cook task tree model depicted in figure 8.7), the object inheritance annotation helps to identify concepts that are spread over the whole application (like for instance the *selectRecipe* object) when modeling complex applications, as well as to identify local concepts (like the *recipeList* and the appliance objects). During the task and object design of the cooking assistant we were able to carefully separate the different levels of abstractions describing a recipe. The recipe is first introduced in the cooking application as part of a database (*recipeList*) in the *findRecipe* sub-tree. The *findRecipe* sub-tree results in a selected recipe (*selectedRecipe*) in the *searchRecipe* sub-tree and then is further detailed in the *Cooking Aid* sub-tree conforming to the tasks to include the description of the preparation steps (in the *Directions* task). When accessing the preparation steps for the user only the active (*currentStep*) and the corresponding next step (*nextStep*) are modeled as relevant. The *currentStep* is required for retrieving the step description and ingredients, as well as the associated appliances and help functionality.

7.4 Domain modeling

We start the domain modeling by collecting all objects that have been identified during the analysis and task modeling phase. For the domain modeling we switch from a user's

7 The Cooking Assistant Case Study

perspective to a system perspective and concentrate on those objects first, that are annotated to all application tasks. Often development is based on pre-existing backend services and therefore application objects already exist that describe the domain data required or produced by pre-existing services. For domain modeling, we identified the following application objects that are listed in listing 7.2:

application object	corresponding tasks
recipeList	loadRecipes(C), filterRecipes (R)
recipeFilter	filterRecipes (R)
SHAsuggestion	loadRecipes (C)
TVsuggestion	loadRecipes (C)
filteredRecipes	filterRecipes (C)
selectedRecipe	manipulateShoppingList (M)
currentStep	manipulateIngredientsAmounts (M), retrieveAppliance (R)
persons	manipulateIngredientsAmounts (R)
applianceList	retrieveAppliance (C)
selectedAppliance	powerOn (M), powerOff (M), waitForCountdownFinished (R), notificationAskForPowerOff(R)

Table 7.2: Application objects and corresponding tasks.

For the cooking assistant application pre-existing services included the appliance control as well as the video-based help that could be accessed by using the Universal Plug and Play (UPnP) protocol. Thus, we re-used already existing services for the *retrieveAppliances*, *powerOn*, *powerOff*, *waitForCountdownFinished*, and *notificationAskForPowerOff* application tasks. Therefore the application objects of these tasks have to be re-used as part of the domain model too.

Since no recipe database existed that structures the recipe content based on certain steps that contain references to useful appliances, we implemented the recipe database on our own. For the cooking assistant case study the recipe database has been realized by using ontologies and JIAC IV services to flexibly choose and plan the order of the preparation steps of a recipe. The planning of the preparation steps was based on the actual appliances that have been connected in the kitchen. Therefore a preparation step contained a pre-condition specifying for instance which appliances are required to support the preparation, and an effect that specified the outcome after a certain step has been performed successfully.

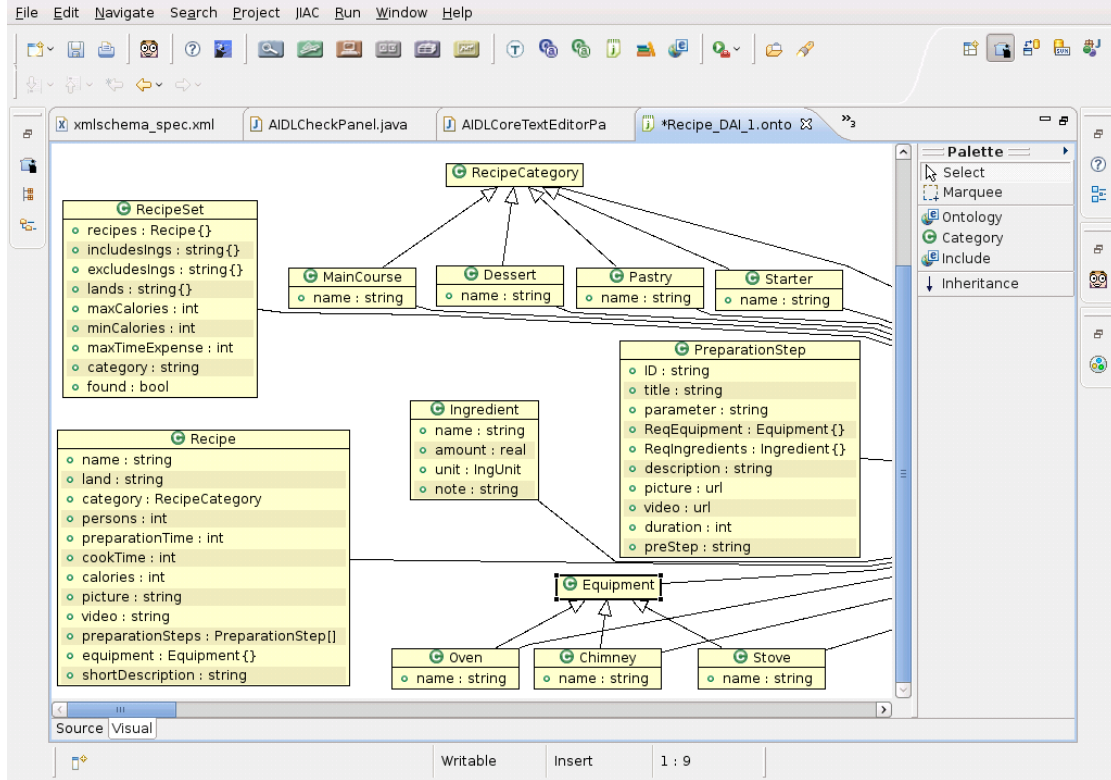


Figure 7.4: An excerpt of the ontology for the cooking assistant.

Figure 7.4 presents parts of the ontology for the cooking assistant including a recipe, which consists of an ordered list of steps, each associated with a list of ingredients, a set of appliances, and a video presentation. Since the recipe structure has been realized in form of an JIAC ontology [Sessler and Albayrak, 2001], we could use the JIAC IV Ontology Editor [Tuguldur et al., 2008] for domain modeling. Looking at the application object list in table 7.2, this required us to implement the *loadRecipes*, *filterRecipes*, *manipulateShoppingList*, and *manipulateIngredientsAmounts* on our own as part of the service modeling that is described in section 7.8.

Domain modeling test step

The testing step during the domain modeling requires deriving exemplary objects based on the scenario description to test the completeness of the domain model structure. Since we re-used parts of the domain model for the UPnP services, for the cooking assistant only the domain model part describing a recipe has to be tested for completeness. Therefore we entered a set of exemplary recipes based on the structure of our recipe database domain model to check, if all information that are relevant for describing a recipe can be captured by our model. We used the Protégé editor [Gennari et al., 2002], which is able to derive a fact editor from our recipe ontology to enter a recipe conforming to our

domain model structure more comfortably.

Listing 8 An excerpt from the lamp chops recipe.

```

1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <recipe>
3   <calories>500</calories>
4   <category> <name>Hauptgericht</name> </category>
5   <cookTime>40</cookTime>
6   <preparationTime>25</preparationTime>
7   <shortDescription>Lammkoteletts mit Knödeln und Gemüse</shortDescription>
8   <land>Deutsch</land>
9   <name>Lammkoteletts</name>
10  <persons>4</persons>
11  <preparationSteps>
12    <preparationSteps>
13      <description>Fetten Sie eine Pfanne mit Öl ein. Würzen Sie dann das Fleisch
14 mit Salz und Pfeffer und reiben Sie es mit zerdrücktem Knoblauch ein.</description>
15      <requiredIngredients>
16        <reqIngredients> <amount>4</amount> <name>Lammkoteletts</name>
17          <unit><name>Stück</name></unit>
18        </reqIngredients>
19        <reqIngredients><amount>2</amount> <name>Olivenöl</name>
20          <unit> <name>tl</name> </unit> </reqIngredients>
21        <reqIngredients> ... </reqIngredients>
22      </requiredIngredients>
23      <picture>Lammkoteletts_0.png</picture>
24      <stepID>FleischVorbereitung</stepID>
25      <title>Pfanne und Fleisch vorbereiten</title>
26    </preparationSteps>
27 ...

```

Listing 8 shows an excerpt of an exemplary lamp chops recipe that we described based on the recipe ontology to test the domain model completeness. The exemplary recipes will be used later on again when rendering exemplary user interfaces based on the abstract and concrete user interface models that are described in the following sections.

7.5 Abstract User Interface Modeling

Complementary to the domain modeling, during the abstract user interface modeling, not the application objects, but the interaction objects are collected from the task model annotations. All abstract interaction objects (AIO) are derived from the application objects of the domain model. The derivation involves two major activities: First, the selection activity to identify the application object data that is relevant for the user and second, the context enrichment activity to help the user to identify the meaning of the selected data.

Application object	Abstract interaction object	Corresponding tasks
Recipefilter	Recipefilter.category	selectPastry (M), selectDessert (M), selectMainDish (M), selectStarter (M), selectLowFat (M), selectMedium (M), disableCaloriesFiltering (M)
	Recipefilter.ingredients.name	removeIngredient (M), addIngredient (M)
filteredRecipes	filteredRecipes.name	listRecipes (R)
SHASuggestion	SHASuggestion.name	listRecipes (R)
selectedRecipe	selectedRecipe	chooseRecipe (C), presentRecipeDetails (R), ingredientIsAvailable (M), IngredientIsNotAvailable (M)
currentStep	currentStep.Title	listRequiredIngredients (R),
	currentstep.description	stepDescription (R),...
applianceList	applianceList.name	selectAppliance (R)
selectedAppliance	selectedAppliance.power	selectAppliance (C), switchOn(R), switchOff(R), presentCountDown(R), notificationAskFor- PowerOff(R)
...		...

Table 7.3: An excerpt of the list of application objects that are selected as relevant for the user (to form abstract interaction objects) and their corresponding interaction tasks that perform on these objects. After each task the object operation is annotated: (R)-Read, (M)-Modify, (C)-Create.

Before the selection activity we require a list of all interaction tasks and their corresponding application objects, which is depicted in the first column of table 7.2. For all these objects we require to derive abstract interaction objects, which we describe in the following two sub-sections.

7.5.1 Selection activity

After all interaction tasks and corresponding application objects have been collected, relevant parts of the objects' data structures for the user interface have to be identified,

7 The Cooking Assistant Case Study

separated and structured into those portions that can be handled by the user interface by performing the selection activity. For the cooking assistant, we use the MASP Builder to derive abstract interaction objects from the collected application objects.

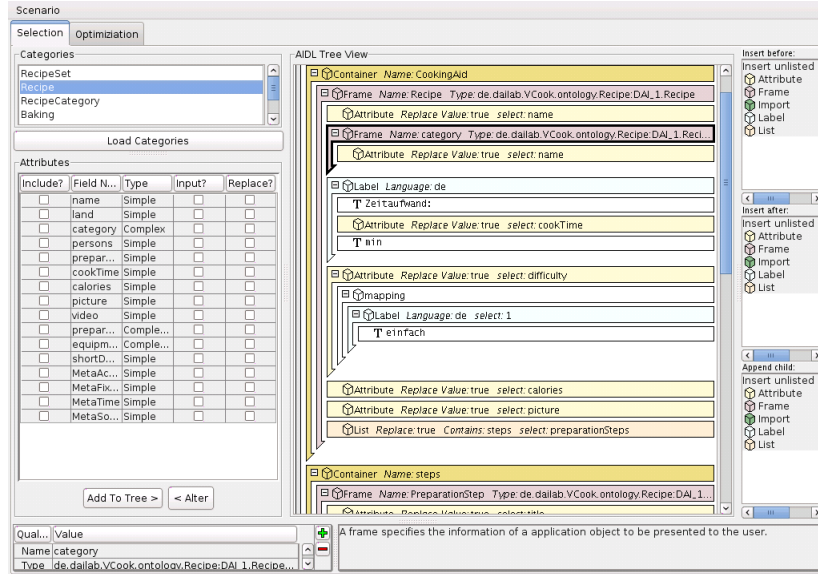


Figure 7.5: A screenshot of the MASP Builder with loaded recipe ontology.

Like depicted in figure 7.5, using the MASP Builder we can load the complete recipe ontology and start the selection activity by choosing relevant data from the ontology. For the cooking aid presentation (see figure 8.4) this includes the basic recipe information like the recipe name or the amount of calories and a picture, the sequence of preparation steps including the ingredients and useful appliances to support the cooking step.

The selection of the relevant attributes of a recipe is supported by the left column of the MASP Builder (see figure 7.5) and can be done by the following subsequent steps: First, we select the recipe from the categories list containing all loaded ontologies that have been designed or reused for the domain model. Next, we select all relevant attributes, and decide if the attribute's data should be initially read from the domain object and presented in the user interface (by the replace flag) and select which of these attributes should be the destination for a user input (by the input flag). Depending whether the selected attribute type is primitive (like for instance strings or number) or complex (for instance an aggregation of another category or a list of aggregated objects) the MASP Builder generates an attribute respectively a frame statement for the AUI model. The AUI model with all selected attributes is visualized in a hierarchy and depicted in the middle row of the MASP Builder user interface like shown by figure 7.5.

In case the designer selects a complex attribute in the MASP Builder, the tool directly adds a new frame to the frame that corresponds to the attribute's category in the AUI model. An example is the selected "category" attribute of the recipe. If the complex attribute contains a list of objects the list element is used, which points to a container.

An example is the recipe's "*preperationSteps*" attribute which we select using the list element that points to the "*steps*" container. Whereas in our case the container "*steps*" stores only one possible set of selection statements describing the data to be presented for each preparation step, it can also contain various compatible selections for a frame. For the cooking assistant we were able to specify the AUI selection to consider the inheritance relationships of the domain model. Since in the domain model all appliances are inherited from a PowerSwitch category conforming to the standardized UPnP service, inherited categories can be selected differently by the AUI selection. Thus, for instance by specifying a basic AUI selection for a PowerOnOff category ensures that the final user interface always supports switching appliances on and off. As long as a new appliance is inherited from the PowerOnOff category the final user interface will support it. Further appliances can be easily considered by adding more specific selections for instance to support an oven, a stove or the airing, each containing further attributes to observe and control appliance specific functionalities.

Beneath the automatic grouping of selected attributes into frames that correspond to the originating category (like for instance the recipe group containing attribute selections for instance the recipe's *name*, *category*, *cookTime*, *difficulty*, *calories*, *picture*, and *preperationSteps*), containers can be introduced to organize sets of selections into a library. In this way frame-based selections for a single application object can be distinguished for several contexts. In the cooking assistant, a recipe is used in various contexts, for instance during the recipe search, we require just the recipe name, its duration and a short description, whereas the cooking aid presentation offers a detailed presentation of the recipe's data. Therefore the detailed selection is organized into a "cooking aid" group whereas the brief recipe description is grouped under a "recipe search" group. All these AUI selections describe the derivation of application objects into interaction objects in several contexts and are collected in a hierarchical AUI model that can be referenced and re-used for various presentations.

7.5.2 Context enrichment activity

After selecting the application object attributes to form interaction objects they have to be put in a context that is understandable by the user. On the interaction object level adding labels to all of the attributes in a language that is understandable by the user is the most important context enrichment action. Without a label for instance the preparation times as well as the calories amount of a recipe are just numbers. Therefore by adding a label statement such as for the "*cookTime*" attribute in figure 7.5, which states the meaning of the integer: "*Zeitaufwand*" (engl.: "Duration:") and the measurement unit "min" for minutes adds a context of the attribute's data that is understandable by the user.

Beneath adding textual descriptions and labels, a basic order of the selected attributes can be set. Thus in figure 7.5 for the cooking assistant, we start by stating the recipe name, followed by the category, preparation time, difficulty, calories, and a picture and thus stating the most relevant information first. If more complex and nested application objects are used within the domain model by applying the container-based grouping,

several frame-based selections can be combined together and organized in an order that is suitable for presenting the interaction objects to the user.

Finally type mappings can be specified for every selected attribute. Type mappings are required to transform attribute values to application objects that are optimized for efficiently being processed by a machine in a human readable format. For the cooking assistant, we required such a type mapping for presenting the level of difficulty to the user. Since the application object attribute uses numbers to store the level of difficulty, we defined a type mapping for instance for the “difficulty” attribute of a recipe stating that a difficulty value of “1” should be mapped to “einfach” (engl. “easy”). Within a mapping for an attribute a set of labels can be defined. In each label element the “select” attribute specifies the source value.

The interaction object modeling ends up with a set of interaction objects that are linked to the corresponding object annotations of the tasks in the task model.

Abstract Interaction Objects modeling Testing Step

With the help of the exemplary objects that have been defined within the testing step of the domain model generation, the abstract user interface can be tested to contain all relevant information in its presentation as well as clearly expressed context enrichments to be easily understandable. Therefore the MASP Builder can load the exemplary objects and generates a generic user interface presentation of the abstract user interface fragments considering all selection and enrichment activities.

Listing 9 Excerpt of the exemplary lamp chops recipe interaction object in an XML-based presentation format.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2   <container name="CookingAid">
3     <frame name="Recipe" type="de.dailab.VCook.ontology.Recipe:DAI_1.Recipe">
4       <attribute replace="true" select="name"/>
5       <frame name="category" type="de.dailab...Recipe:DAI_1.RecipeCategory">
6         <attribute replace="true" select="name">
7           <value>Lammkoteletts</value>
8         </attribute>
9       </frame>
10      <label lang="de">Zeitaufwand:<attribute replace="true" select="cookTime"/>
11      <value>40</value>min</label>
12      <attribute replace="true" select="difficulty">
13        <value>mittel</value>
14      </attribute>
15      ...
16      <list contains="steps" replace="true" select="preparationSteps">
17        <frame name="PreparationStep" type="de...Recipe:DAI_1.PreparationStep">
18          <attribute replace="true" select="title">
19            <value>Pfanne und Fleisch vorbereiten</value></attribute>
20          <label lang="de">Gerätesteuerung<list contains="appliances"
21          replace="true" select="ReqEquipment">
22            <frame name="PowerControl" type="de...SHS_Device:DAI_1.PowerControl">
23              <attribute replace="true" select="Power">
24                <value>off</value></attribute>
25              <attribute replace="true" select="description">
26                <value>Oven</value></attribute>
27            </frame>
28          </list>
29          </label>
30          <label lang="de">Zutaten<list contains="ingredients"
31          replace="true" select="ReqIngredients">
32            <frame name="Ingredient" type="de...Recipe:DAI_1.Ingredient">
33              <attribute replace="true" select="amount">
34                <value>4</value></attribute>
35              <frame name="unit" type="de...Recipe:DAI_1.IngUnit">
36                <attribute replace="true" select="name">
37                  <value>Stück</value></attribute>
38              </frame>
39              <attribute replace="true" select="name">
40                <value>Lammkoteletts</value></attribute>
41            </frame>
42          </list>
43          </label>
44          <attribute replace="true" select="description">
45            <value>Fetten Sie eine Pfanne mit Öl ein. Würzen Sie dann das Fleisch
46 mit Salz und Pfeffer und reiben Sie es mit zerdrücktem Knoblauch ein.
47          </value></attribute>
48          ...
49        </frame>
50      </list>
51    </frame>
52  </container>

```

7 The Cooking Assistant Case Study

Listing 9 presents the XML presentation of the exemplary lamp chops recipe interaction object. Based on this XML presentation of the exemplary interaction object, by using the MASP Builder, we can generate generic user interface renderings for a visual (HTML) and a textual (voice) representation.

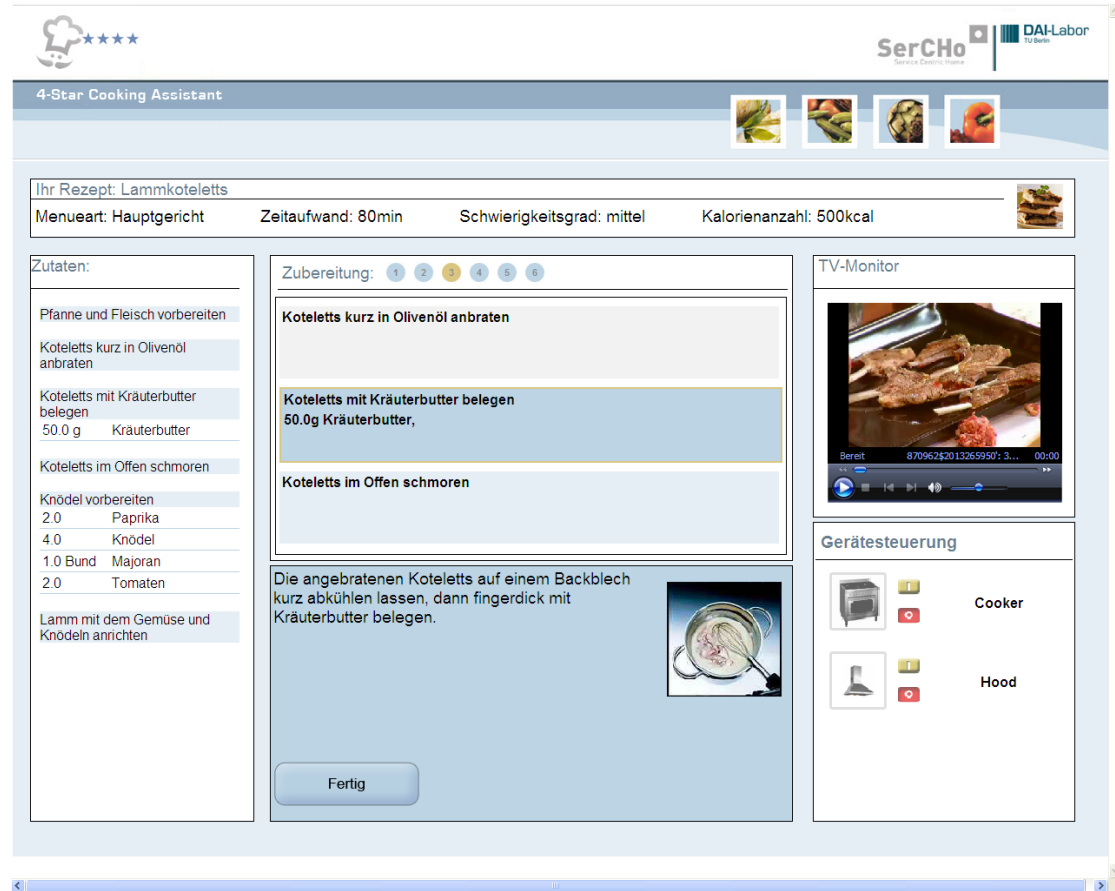


Figure 7.6: The HTML presentation of the lamp chops interaction object in a browser.

The screenshot in figure 7.6 depicts such a generic display using the example lamp chop object. The presentations of the abstract user interface objects based on the exemplary objects can be generated as a visual (HTML) and a textual (voice) representation by the MASP builder. Whereas testing the visual representation typically results in shorter texts to reduce the presentation space, or more detailed descriptions to support the user in complex tasks, testing the speech presentation of the user interface helps to improve the wording and reducing the complexity of longer texts.

7.6 Layout modeling

After the abstract user interface objects have been derived, the graphical user interface layout can be derived for the cooking assistant by following a model-based layout generation. For the cooking assistant a flexible user interface layout should be realized allowing to adapt the user interface to several different screen sizes starting from mobile devices enabling the user to transfer the cooking assistant to his mobile device, to wall mounted touch screens up to a huge LCD-television that is installed in the user's home environment. Further on, based on contextual information like for instance the user's distance to a device the user interface layout should be adapted at run-time.

The MASP layout model generator helped us to systematically derive user interface layouts that are consistent to the other design models. Therefore we can load all design models into the layout model generator and define layouting statements that specify model interpretations resulting in layout decisions for the concrete user interface. As layout statements can be defined context sensitively, the specification of layout adaptations to support a specific platform (for instance a touch screen) or the environment (for instance the user's distance to a display) is possible.

As described in section 4.2.5, we distinguish four different kinds of statements in the model-based layouting, each addressing a specific layout characteristic: The containment, the order, the orientation, and the size of the user interface elements. For the cooking assistant we started by defining the containment structure.

The Layout Generator gets initiated with default statements. First, by a containment-related statement we derive the initial containment structure from the task model. Therefore the statement traverses the task tree and generates a container for each abstract task and container elements for atomic tasks. Next, by an orientation-related default statement we toggle the orientation from horizontal to vertical and vice versa traversing the containment structure from the overall container to the atomic elements. Finally a size-related statement defines an initial height-to-width size relation for individual elements to meet the factor 1:2. All statements can be replaced during the layout modeling process but ensure that an initial layout can be generated.

7 The Cooking Assistant Case Study

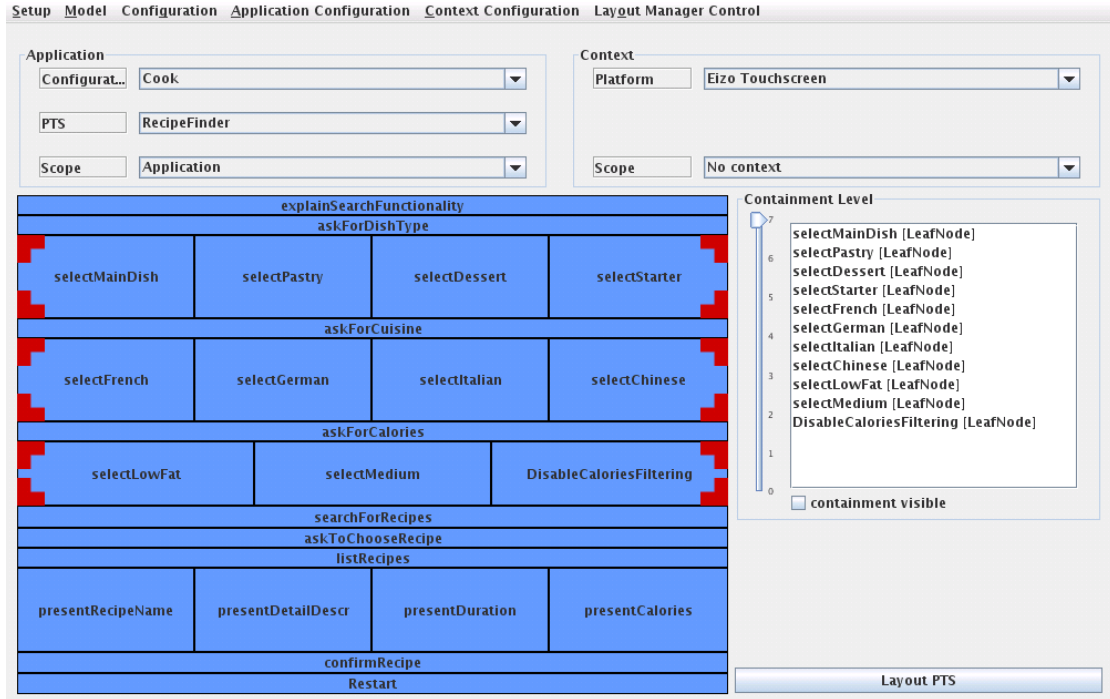


Figure 7.7: The initial layout of the recipe finder screen as it is generated by the layout model generator.

Figure 7.7 depicts a screenshot of the box-based layout preview for the Recipe Finder screen of the cooking assistant (figure 8.2). By the containment slider we can browse through the nesting levels. Like depicted in figure 7.8, we defined a new container “dishtype0” and two containment statements that state that the elements “selectStarter” and “selectPastry” have to be children of this container. Because of the initial orientation toggling statement, both elements get instantly orientated vertically to each others. As all four elements that can be used to filter the recipe database regarding the dish type have been originally defined to share the same parent task, the definition of a new container inserts a new element grouping to be relevant only for the user interface layout.

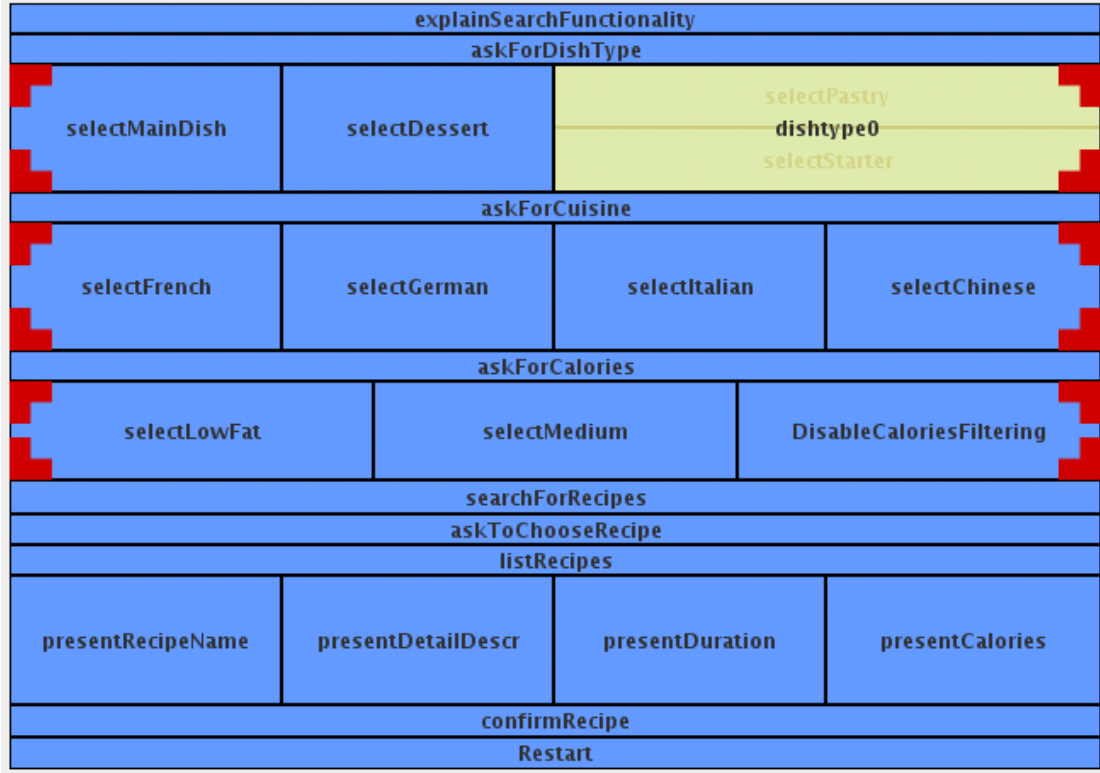


Figure 7.8: We change the containment structure that has been derived by the task model and introduce a new container “dishtype0” that contains the “selectStarter” and “selectPastry” elements.

After the containment-related statements have been defined, order-related statements are used to define the element order for elements in the same container. By using the containment slider in the Layout generator we can easily check the containment relationship and identify those atomic elements and containers that share the same parent container. For the cooking assistant we want the restart element to be always the first element to navigate to. Thus we can use the containment slider to navigate to a containment level in that both the restart element and the container that contains the rest of the user interface (the “cooking application” container) share the same parent container (“StarCook”). By using a context-popup menu over the “CookingAppliction” box we can define a new order related statement that describes, that “Restart” has to come before “CookingApplication”. Defining such an order related statement results in an updated presentation that is depicted in figure 7.9.

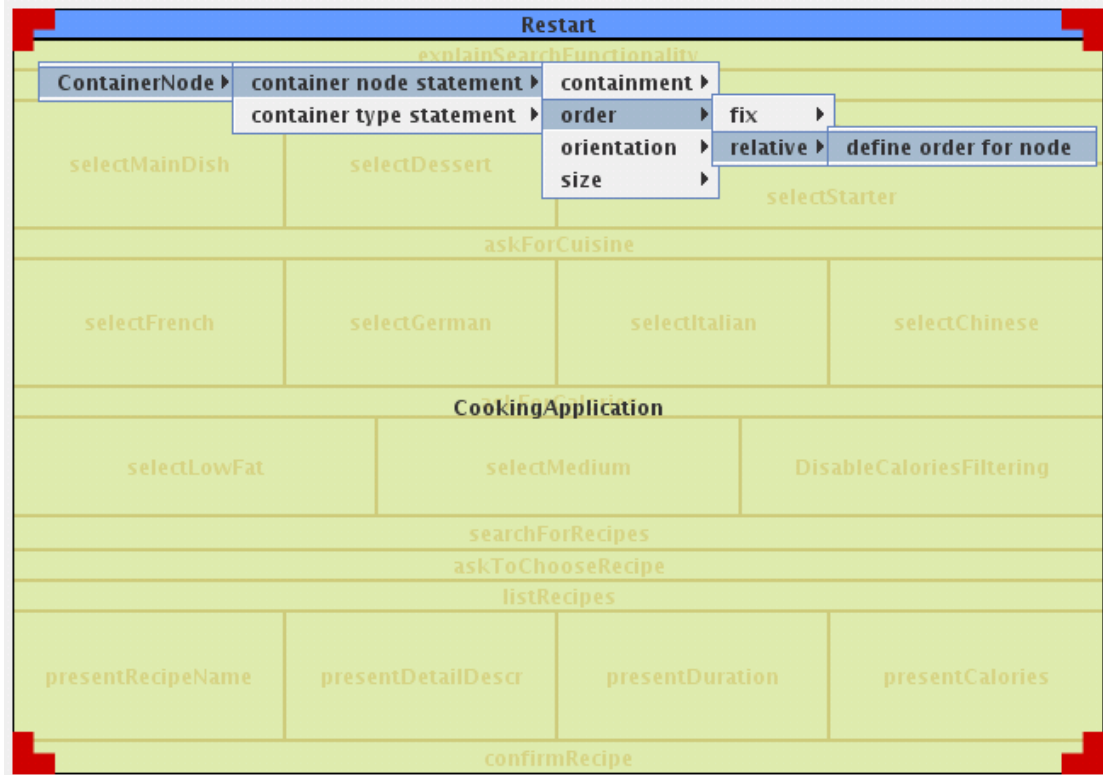


Figure 7.9: The layout of the recipe finder screen after the addition of an order-related statement.

Now the Restart element is the first element to be displayed respectively the first element that would be read when using a speech-based user interface. Whereas the containment and the order statements are relevant for both, visual and speech-based user interfaces, the orientation and size criteria are only relevant for visual interface layouts. Both orientation and size statements help to optimize the usage of the display space on a graphical output device.

An example for an orientation-related statement that we used for the recipe finder screen is the decision for a two column layout that we used to visually separate the criteria search form (“refineSearch”) from the list containing all recipes that matched the search criteria (“browseRecipes”). By using the containment slider and browsing to the parent container (“RecipeFinder”) that contains both elements, we can define horizontal orientation statements for all elements of this container and therefore overwrite the default orientation toggling for these elements. Figure 7.10 depicts the resulting layout after the definition of the orientation statement.

7 The Cooking Assistant Case Study

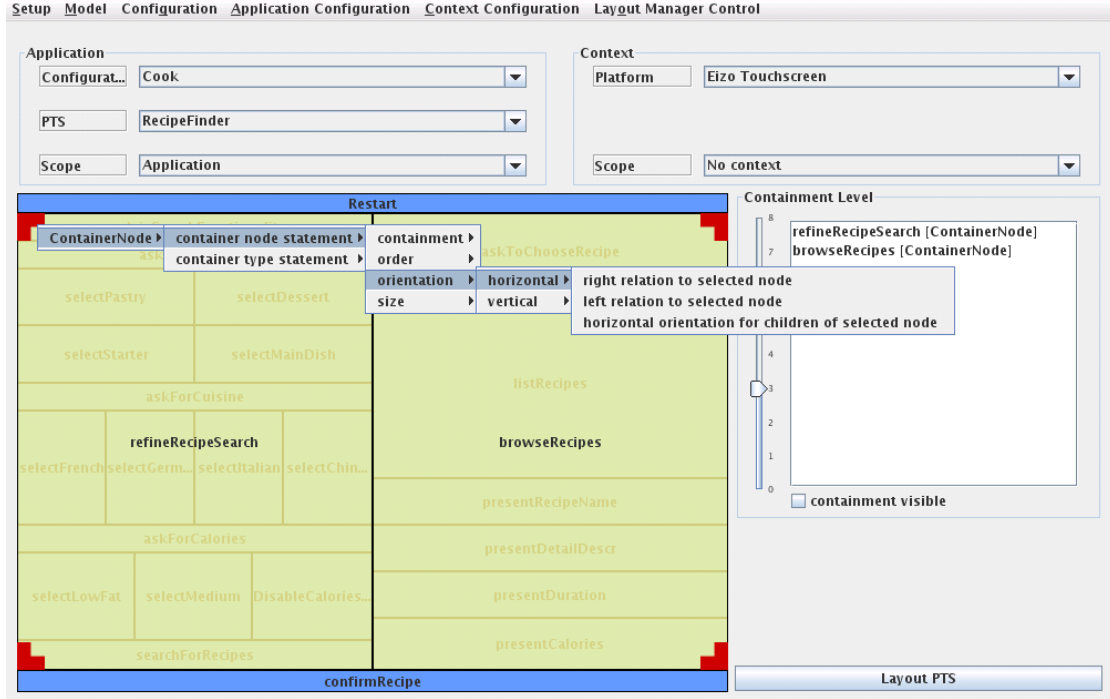


Figure 7.10: The layout of the recipe finder screen after the addition of an orientation-related statement.

Finally size-related statements can be specified to manipulate individual elements' or containers' sizes to adapt to individual display sizes of the supported target devices. Beneath linearly scaling all elements to match a specific display, the element scaling can be defined separately for different contexts-of-use. For the cook case study, we required a layout adaptation to a wall-mounted touch screen. For this platform we specified separate size-related statements defining that all elements that will be defined as a button in the concrete user interface model need to be presented with at least 50x50 pixels to be easily controllable with a touch screen. Additionally, we can specify a size-related statement considering the distance from the user to the display. For the cooking assistant on the touch screen we configured that all elements that have been modeled as output elements in the abstract user interface model should be magnified as the user moves away from the display. Since all statements addressing a specific layouting characteristic are set up in a strict order that reflects their evaluation priority, we ensure for instance that both the minimal button size (a button corresponds to an input element in the abstract user interface) and the distance adaptations do not restrict each others.

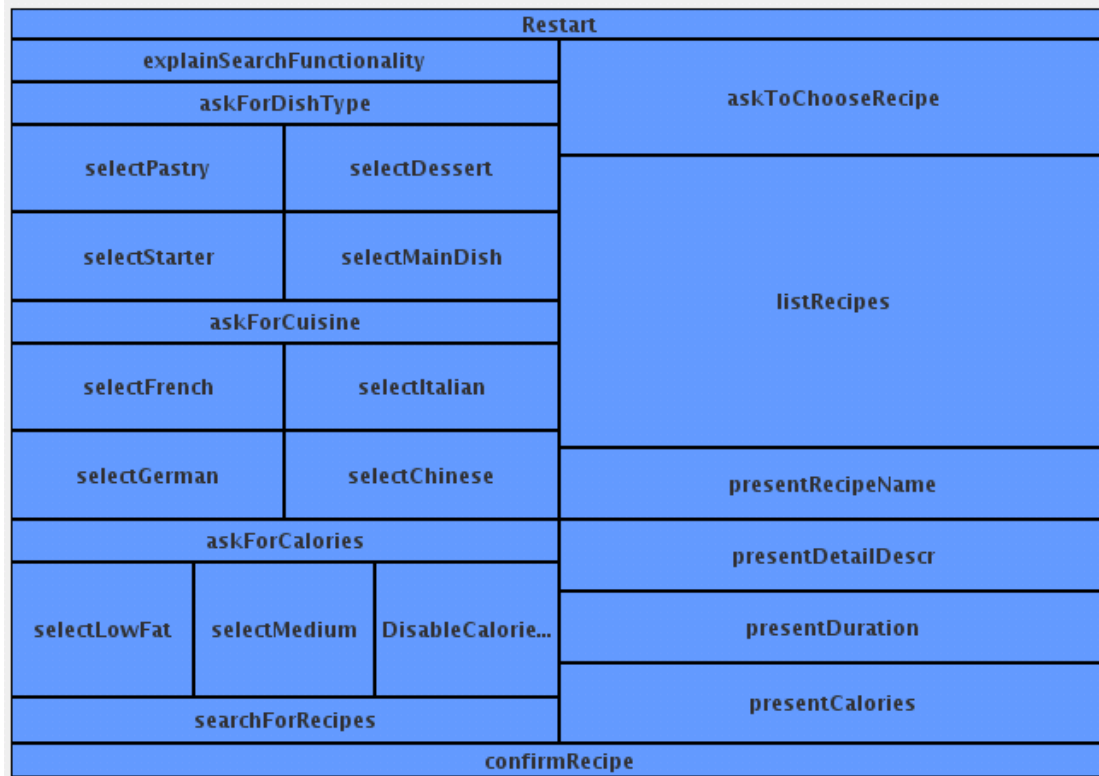


Figure 7.11: The final layout of the recipe screen after all statements have been defined.

Figure 7.11 illustrates the final layout preview of the layout model generator for the recipe finder screen, which required us to define 9 containment, 3 orientation, 3 order and 4 size-related statements.

Layout modeling testing step

By using the boxed-based layouting preview in the layout model generator tool, the designer receives instant feedback for every new statement that she adds to the layout model. Instead of just previewing boxes, each box can be replaced by a picture that depicts the content of the user interface fragment to give the designer a better impression, about the consequences when resizing a box. These pictures can be cutted out from picture-based user interface mockups or can be retrieved from a library that stores pictures for common user interface widgets. During working with the picture-based preview of the tool, we figured out that expecially for defining size-related statements the pictures support specifying correct aspect ratios (the tool can automatically define a statement corresponding to the aspect ratio of a picture). Additionally if boxes contain textual descriptions, a picture illustrating the text helps to identify the minimum size of a box and ensures that the text remains readable after layout adaptations.

Statements can be defined on an global application level as well as for a specific screen

or a certain platform or environment condition. Therefore the tool enables the designer to browse through all the screens of the application as well as to the various contexts-of-use that should be considered by the applications in order to check if the layout model modifications have been considered correctly.

Further on, the tool can be directly connected to a running MASP environment, which has all the cooking assistant models loaded and instantiated. In this case the designer can commit his changes to the layout model directly to the running application to prototype layout adaptations by real world scenarios. This makes especially sense if context-of-use adaptation to changes in the environment (like the distance-based adaptation) should be tested.

For the cooking assistant case study we also tested the design efficiency of our model-based layouting approach, which will be presented in the next sub-section before we continue describing the final user interface generation.

Evaluation of the design and run-time efficiency

We tested our approach regarding two aspects: First, the efficiency at design-time for the designer to generate the layout model by using the layout model generator. Second, we tested the efficiency of the implementation to generate and solve the constraints in our run-time system.

To test the design efficiency of the approach, we asked a designer to realize a layout for an interactive cooking assistant application based on a textual description of a scenario of how the cooking assistant should support the user. The designer created three screens and one user interface distribution scenario where one screen is split to two different devices: The initial screen asks the user to search for a recipe based on several search options. The second screen is about assisting the user to generate a shopping list by asking the user which of the required ingredients are available and which are not available. This screen could be split into two parts where one part gets distributed onto a PDA that could be taken along during shopping and the other part remains on a touch screen in the kitchen. The last screen assists the user during cooking by offering video-based help, controlling the kitchen appliances and by splitting each recipe into a list of steps containing the required ingredients as well as a detailed description about what to do in each step. Figure 7.12 presents the initial screen for the recipe search as it has been realized by the designer and the result of the model-based layouting using the box-based layout of the editor.

7 The Cooking Assistant Case Study

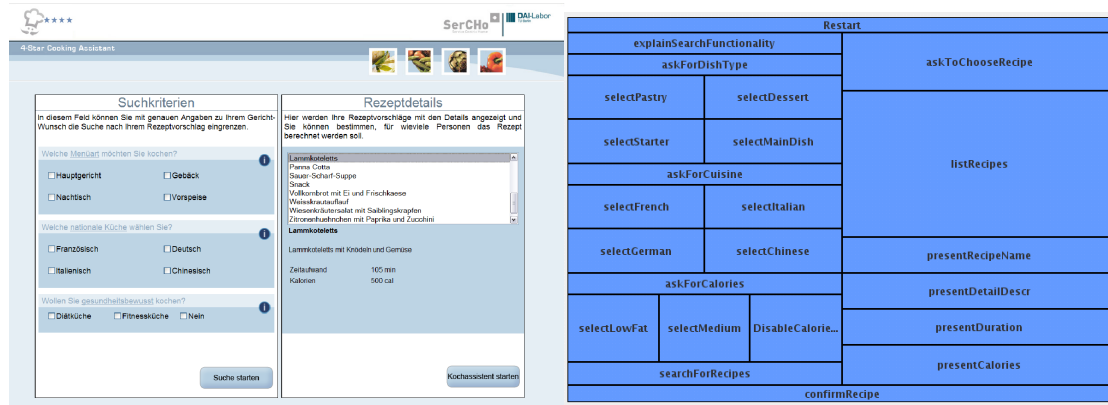


Figure 7.12: The screen for the recipe search and the final box-based layout result of the layout-model generator

Independently from the designer we asked a developer to follow a model-based development approach. Initially both, the designer and the developer shared the same textual description of a scenario for the cooking assistant. Based on the results of the model-based development approach including a fine grained task model, a domain model and an AUI model, we then derived a model based layout that should correspond to the screens of the designer as close as possible. Finally, we measured the number of statements that have been required to end up with the same layout as the designer has proposed for the cooking assistant.

Each screen has a different layout complexity consisting of a number of elements that are nested based on the abstraction levels of the task model.

Screen	1)	2)	3)	4)	5)	6)	7)
1. Recipe Search	19	7	9	3	3	4	19
2. Shopping List	13	8	0	6	1	2	12
3. Distribution: PDA, Touch	4,9	8	0	1,2	0	0	0
4. Cooking Aid	15	10	0	2	2	4	8

Table 7.4: Complexity of the screens that we sequentially layouted one after another and the number of statements required. 1) Elements to layout, 2) Abstraction levels, Number of 3) containment-, 4) orientation-, 5) order-, 6) size-related statements. 7) Total number of statements (additionally statements required to layout a certain screen only - statements of previously layouted screen are reused).

Table 7.4 lists the level of complexity (number of elements, and the maximum nesting level utilized) for the three screens that have been sequentially layouted and the number of statements that were required to realize the layout of the designer. After the first screen has been layouted the derived statements have been re-applied to the second screen and finally to the third screen. The second column of table 7.4 lists the different levels of

complexities that we have considered by the three screens: Whereas the RecipeFinder screen has a lot of elements (19) and a less nested structure of 7 levels, the Cooking Aid screen has 15 elements on 10 nesting levels as it is composed of various parts that are not directly related (for instance the multi-medial help and the appliance control). For the ShoppingList screen two further layouts have been designed that are reflecting a distribution scenario where parts of the screen get distributed to a PDA (4 elements) and some parts (9 elements) remain on the wall-mounted touch screen in the kitchen. By analyzing the amount and type of statements that were required to layout the screens in the same way like the designer did, several observations have been made:

- Containment and order related statements can be derived from a task tree efficiently.
- If the task model is used to derive the containment and atomic tasks are identical to individual widgets, the introduction of further containment-related statements is required (for our application we required 8 containment statements for grouping checkboxes for the recipe search screen).
- Size-related statements can be defined very efficiently on an application wide, global level based on the information of the design models (such as weighting input to output tasks, or by giving control tasks that usually end up presented as buttons a global minimum /maximum size restriction).
- The aspect ratio has to be defined for each individual picture that should be presented within a task (using a size relational statement) which can be automatically derived at run-time when loading the picture.
- The orientation-related statements can only be specified on a global level but have to be re-applied for most of the individual screens. This is because the user interface models have no information that can be used to derive an initial orientation. So we applied a heuristic approach that produces elements with a balanced width-to-height relation by switching the orientation of the elements. Therefore we toggle the orientation horizontal to vertical and vice-versa, for each nesting level that has been derived from the task model.
- The container, order- and size-related statements of the layout model helped to assemble layouts for user interface distributions that have not been explicitly addressed at design-time. Orientation-related statements caused problems as after a distribution has been initiated by the user the re-orientation of the remaining user interface parts were not expected by the users.

In order to check if the performance of generating and solving the constraints can be done at run-time like presented in section 5.2.4, we measured the performance of both, the statement evaluation and the constraint solving separately.

7 The Cooking Assistant Case Study

Screen	1	2	3	4
1.Recipe Search	25	142	<1ms	14 ms
2.Shopping List	20	107	<1ms	8ms
3. Distribution: PDA, Touch	8,10	56,81	<1ms	8,10ms
4.Cooking Aid	23	130	<1ms	13ms

Table 7.5: Complexity of the screens the need to be layouted and the amount of statements required.1) Number of statements to evaluate, 2) Number of evaluated constraints, 3) Measurement for statement evaluation (ms),4) Duration for constraint solving (ms).

Table 7.5 shows the results of the performance evaluation for our cooking assistant application. For each screen we have measured the amount of statements that have been selected as relevant for layouting each screen (second column) and the amount of constraints that have been generated by evaluating the selected statements. It could be observed that currently in the average 5 to 7 constraints are generated by one statement. In the last two columns the measured average calculation time (of three runs using an Intel Core Duo L2400 processor with 1,8 GHz) for selecting the required statements and the duration for solving the generated constraints are listed: We could observe that the time to choose between the statements that are relevant for a specific situation was always under 1ms. The amount of constraints that are generated by the selected relevant statements influences the solving time. Thus, the bigger the difference between the overall number of layouting statements and the number of selected statements, the shorter constraint solving times can be expected.

7.7 Concrete user interface modeling

By the concrete user interface modeling the abstract user interface, which is designed modality independent, is transformed to a set of modality specific user interfaces. In our approach we experimented with concrete user interface generation for HTML, WML, and Thinlets to support desktop PCs, small-sized and mobile devices, and voice in/output generation relying on VoiceXML.

Similar to approaches like UsiXML [Limbourg and Vanderdonckt, 2004a], TERESA [Mori et al., 2004] or Dygimes [Coninx et al., 2003], we utilize XSL transformations to generate the final user interface, but instead of directly applying transformations to derive the CUI from the AUI, we follow a multi-level transformation approach.

For the cook case study we used multi-level XSL transformations. Therefore three different kinds of transformations are required to derive a concrete user interface: For each modality that should be considered, a generic transformation generates a specific transformation that is able to transform the abstract user interface into a concrete user interface model for a specific device. The specific transformation can be extended to include design-related information considering design-guidelines or specific style guides. The multi level transformation approach ensures that each transformation follows a basic

structure and eases the extensibility to include new user interface elements. Further on it reduces the effort for the developer to produce a new user interface since the initial specific transformation to include a new platform can be automatically generated and is ready to produce an initial user interface that can be subsequently improved by the developer instead of requiring the developer to start from scratch. The automatic generation additionally constructs a performance-optimized transformation structure, which is often neglected by the developer because of the initial programming efforts.

The MASP Builder tool includes general transformations to generate specific transformations for VoiceXML, static HTML, Velocity HTML, WML, and Thinlets. Based on the abstract user interface description fragments the tool automatically generates a template for a specific transformation that can be directly applied to the abstract user interface fragment to preview the final user interface.

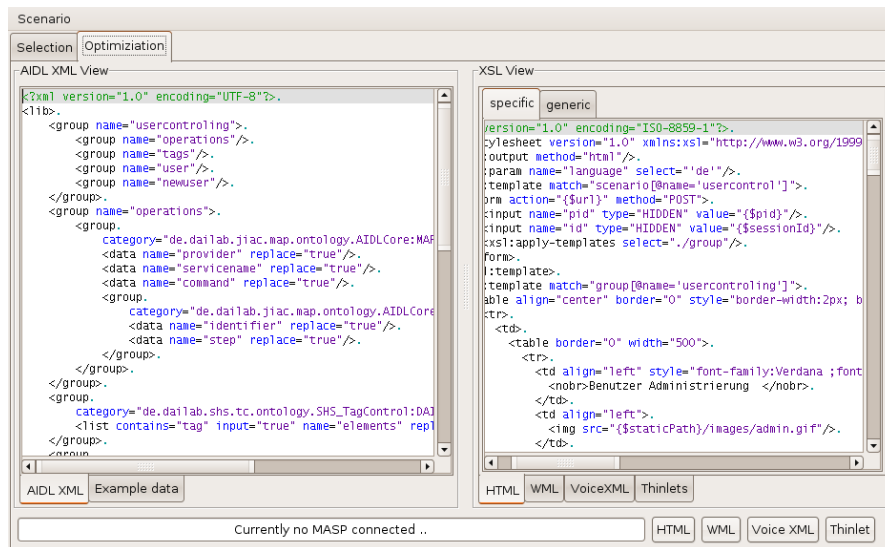


Figure 7.13: Screenshot of the MASP Builder with the specific transformation for the cooking assistant.

Figure 7.13 shows a screenshot of the specific HTML transformation. As it has been automatically generated by the tool, it follows a basic structure: In the first part of the transformation the element order of the abstract interaction object is specified, whereas in the second part the data selection and context enrichment statements have been placed for all interaction object separately. Modularizing the interaction objects and assembling them separately improves the layouting possibilities later on. By deriving the specific transformation the application can be further extended to include specific fonts, button layouts and additional graphics. Additionally interaction object templates of the specific transformation can be overwritten to implement a specific design or a widget for instance.

Using multi-level transformations helped us to reduce the often mentioned critique about the complexity of XSLT by introducing a new level of abstraction, a generic transformation that is used to generate platform specific XSL transformations implementing

strategies to increase the performance of the transformation. The concept of multi-level transformation is based on the idea that XSL transformations are written in XML and can therefore be generated using another more abstract XSL transformation.

For the cooking assistant we implemented a generic transformation to generate specific templates that produce Velocity Templates. A Velocity template is a mixture of HTML or VoiceXML text, which can be embedded with java code and directly compiled within the webserver to produce HTML or VoiceXML code that can be interpreted by a Webbrowser or a Voicebrowser, respectively.

Listing 10 Velocity HTML template for the selectDishType user interface fragment.

```

1 <div id="selectDishType">
2     <div id="selctDishTypeBody">
3         <div class="refineSearchParametersTitel">
4             Welche Nationale Küche wählen Sie?
5         </div>
6         <div class="refineSearchParametersItem">
7             <input id="#arabic" name="kitchen1"
8                 #if($recipe.arabic)checked="checked"#end
9                 type="checkbox" value="Arabisch"/>Arabisch
10        </div>
11        <div class="refineSearchParametersItem">
12            <input id="#german" name="kitchen2"
13                #if($recipe.german)checked="checked"#end
14                type="checkbox" value="Deutsch"/>Deutsch
15        </div>
16        <div class="refineSearchParametersItem">
17            <input id="#italian" name="kitchen3"
18                #if($recipe.italian)checked="checked"#end
19                type="checkbox" value="Italienisch"/>Italienisch
20        </div>
21        <div class="refineSearchParametersItem">
22            <input id="#chinese" name="kitchen4"
23                #if($recipe.chinese)checked="checked"#end
24                type="checkbox" value="Chinesisch"/>Chinesisch
25        </div>
26        <div class="clearboth"></div>
27    </div>
28 </div>

```

Listing 10 presents such a velocity template for the selectDishType user interface fragment that we use to generate HTML code. In the velocity template, each interaction object is embedded into a div-tag with a unique name so that the interaction object can be addressed by the user interface layouting that can set the presentation coordinates as well as the size for each interaction object separately. For each concrete UI interaction object type the transformation generates the corresponding widget for the target platform. Like depicted in Listing 10, we transform the interaction object dishType which has been

defined as a multi-selection list in the corresponding abstract user interface object to a HTML-checkbox widget.

Concrete User Interface modeling testing step

During the derivation and design of the specific transformations for all the modalities and devices that should be supported by the cooking assistant, by using the MASP Builder, we are able to generate instant static previews of the final user interface for all platforms that we want to support. Therefore the tool can connect to several web browsers for previewing the HTML-based designs, WAP and Thinlets simulators as well as connect to a remote voice server that can initiate a telephone call to a phone number that has been configured in the preferences of the MASP builder.

Beneath previewing static designs of certain presentations, the tool is able to connect to a running MASP environment that has been started with all the models of the cooking assistant application. By connecting the tool with the running application that tool can directly update the concrete user interface transformations model of the cooking assistant in the MASP. Through such a model update the presentation of the running cooking application gets instantly updated. Therefore the modified concrete user interface models can be directly previewed and tested in a consistent dynamic and interactive situation that is composed out of several user interface fragments.

7.8 Service Model

The service model is generated based on the information of the task and domain models and connects the functional core to the user interface of an interactive application. Two different types of services must be considered by the service model. On the one hand, already existing or proprietary services and on the other hand new functionalities that have been specifically implemented for the application should be easily integrated by the service model.

Since the cooking assistant should be tightly integrated in a smart home environment, we extensively integrated and implemented both kinds of service types. For the appliance control functionality we were required to integrate an already existing vendor specific API, based on web services. The context sensitive, video-based help in each step of the cooking process was offered by a media center PC with recording capabilities that was able to stream its multi media content via the standardized Universal Plug and Play (UPnP) Media A/V protocol. Finally the recipe database has been set up based on the recipe ontology that has been developed during the case study. All the small functions to calculate the amounts of ingredients based on the number of persons or the generation of the shopping list have been implemented in Java.

Following the basic requirement for separation of concerns that we formulated in section 5.1.3.2, the functional core is required to be loosely coupled to the user interface enabling the developer to easily exchange parts of the functional core to consider a change in technology like for instance connecting appliance from another vendor without requiring the developer to know about the internals of the other models.

Looking at the task tree of the cooking assistant we require service calls to the functional core for all application tasks. By using the Task Model Editor, we are supported in specifying for an application task the underlying technology that should be considered by selecting a suitable service adapter (we have implemented service adapters for UPnP and Java calls). Since for each application task all domain model objects as well as their access operations (read, manipulate) have been already annotated during the task modeling phase, the service model can be automatically generated using a transformation.

Listing 11 An excerpt from the service model of the cooking assistant.

```

1 <bean id="RecipeSearch" class="org.sercho.masp.meta.service.ServiceCallDescription">
2   <constructor-arg><value>RecipeSearch</value></constructor-arg>
3   <constructor-arg><ref local="JavaServiceType"/></constructor-arg>
4   <property name="instruction">
5     <value>finder.getRecipeSet(recipeFilter)</value>
6   </property>
7   <property name="parameterIDs">
8     <set><value>finder</value> <value>recipeFilter</value></set>
9   </property>
10 </bean>
11 <bean id="PowerSwitch" class="org.sercho.masp.meta.service.ServiceCallDescription">
12   <constructor-arg><value>PowerSwitch</value></constructor-arg>
13   <constructor-arg><ref local="CFServiceType"/></constructor-arg>
14   <property name="instruction">
15     <value> PowerSwitchUDN=PowerSwitchUDN ResultName=resultName
16           Observers=org.sercho.controller.bsh.BSHObserver </value>
17   </property>
18   <property name="parameterIDs">
19     <set><value>PowerSwitchUDN</value></set>
20   </property>
21 </bean>

```

Listing 11 shows an excerpt of the service model of the cooking assistant for two services, the recipe search functionality that has been realized in java and the PowerSwitch to switch an appliance on or off by using the UPnP protocol. Whereas the former requests for a list of recipes that matches the criteria that the user specified in the recipeFilter object, the latter identifies an appliance based on the appliance identifier (PowerSwitchUDN) and then switches the power of the appliance.

7.9 Deployment phase

All these Velocity pages are then deployed to the FUI model, together with the task model, the domain model and the service model to generate the final user interface during system run-time. The service signatures of the service model are generated by referring to annotations of the application tasks that are used to specify the service technologies (in our implementation web-services, native Java calls, and UPnP calls) together with

the application objects to transform them into code fragments (via XSLT) that will be called during task execution.

7.10 Conclusions of Case Study

The cooking assistant has been implemented as an exemplary application as part of the SerCHo project, which has been running over three years and involved seven partners from the industry. After each year, an acceptance study has been undertaken by an external company that is specialized on market research [Schäfer, 2007b]. A systematic usability testing approach was beyond the scope of the project, since the studies have been focused on testing the market acceptance of all the prototypic applications of the SerCHo project.

During the user study we evaluated early prototypes of the cooking assistant based on the presented method, the underlying MASP architecture and the cooking assistant demo application. The basis for providing a realistic environment was an Ambient Assisted Living Testbed we set up in close cooperation with the Deutsche Telekom Laboratories at the Technical University of Berlin. Consisting of a living room, an office and a kitchen, the Testbed provides the ideal environment to evaluate smart home applications like the cooking assistant under realistic circumstances.

Based on the results of the acceptance study we iterated the complete method again and continuously improved the cooking assistant based on the users' feedback, which has been gained by a structured questionnaire with 20 persons from different potential target groups and a moderated group discussion with 10 persons afterwards. We have published results of these interviews in [Blumendorf et al., 2007].

Most of the initial feedback we gained by the user study was targeted to the complexity of the graphical user interface, which included too much information especially in the cooking assistant screen. Whereas the first cooking aid screen included a presentation of all steps and a list of all ingredients for a certain recipe, we continuously reduced the cooking aid screen ending up with just presenting one step and give a summary of the next step as well as presenting the ingredients list, the video-based help, and the appliance control context-sensitive to the current step. We improved the method and developed the task simulator based on the lessons learned by the case study. By the task simulator an early feedback can be gained by testing the number of simultaneous presented tasks as well as checking the designed task flow in advance with the targeted audience.

8 Conclusions

8.1 Context of this Work

Model-based transformational development of interactive systems has been continuously undertaken over more than two decades. By applying the concept of transformational development, which has its roots in traditional software engineering, to human computer interaction structured methods based on the specification of models have been successfully developed. Whereas a lot of approaches for methods, tools, and models for the development of user interfaces have been proposed, few of them concentrate on certain aspects like the identification and specification of models, the classification of relevant architectures or the application of model-based user interface development to several application domains like mission critical interactive systems or multi-modal interaction in cockpits of aircrafts or even in cars.

The identification and specification of models is currently focusing on the further formalization of task-models and its relation to the domain and concepts models, the identification of more general process-flow-oriented model descriptions, the specification of dialog models and the development of concrete models describing user interface elements for certain modalities to support or instance haptic interaction or the generation of 3D user interfaces. Up to now a generally accepted description language for the specification of all these models is still missing. Over the last decades UsiXML, UIML, and CTT/TeresaXML are the only languages that have been targeted to continuous further improvements from the research community. UsiXML has been designed as a proposal for a standardized language and currently includes specification for most of the actual identified abstraction models but actually missed a detailed formalization of its semantics and is mostly applied as an exchange format to exchange models between all the tools.

The evolvement of general architectures to describe interactive systems has been most active in the 90s. Nowadays the further evolvement has been replaced by a discussion about frameworks to classify all the architectures that are currently being produced in the various application areas. Only limited effort is currently ongoing to address architectural challenges of model-based application. Architectural challenges include to support all the various abstraction models, allowing to add further models to address specific application domains, as well as propagating changes of individual models to all other connected ones enabling evolutionary prototyping and multi-modal fusion.

There are a lot of application areas that motivate the need for model-based development. Already the overall complexity of actual interactive application demands for a structured development process to ensure the consistency of the user interface for the whole interactive application. This complexity even raises with the development of multi-modal, multi-platform, multi-user user interfaces, as well as user interfaces that can adapt

to various contexts-of-use.

8.2 Content of this Dissertation

The work on this dissertation started with the initial idea of implementing an interactive application by following a model-based user interface development process. During the process of development the developer should be supported to test the actual state of the design as early and often as possible.

The state of the art analysis in chapter 2 revealed several shortcomings and the inspection of the requirements lead us to three main areas of interests that this work is concentrating on:

1. We required a method that is efficiently tool-driven, supports all phases of the development process and supports testing support in all phases to consider feedback as early and often as possible.
2. The need for an architecture arose, enabling us implementing an interactive system based on assembling those abstraction models that we figured out to work best for a certain problem domain. The architecture should allow changes to the already running application such as required to prototype several usage-scenarios, enabling adaptations to various contexts-of-use or even testing the utilization of additional modalities for interacting with the system.
3. The improvement of existing models as well as the addition of further models to support the adaptation of an application to several contexts-of-use and to enable directly executing these models to eliminate the actual gap between design-time and run-time in interactive system development.

Regarding (1), we derived in chapter 4, based on the observations (subsec. 4.1.1) of actual design methods in the Start Of The Art analysis (chapter 3), the requirements (subsec. 4.1.3) that forced our work realizing the MASP Methodology. The basic idea of the method is to structure the phases of a software engineering process into subsequent steps specifying the user activities, which have to be done with support of a tool either in an interactive or an automated way. Different to other methods in model-based development, we enable the developer to test intermediate results in each step of the methodology to check as early as possible for inconsistencies and discrepancies regarding the initial scenarios and system requirements. Therefore exemplary views, simulations or early prototypes can be derived in every step enabling the developer to incorporate user feedback even in the very early states of development to prevent defective developments caused by misunderstandings with the targeted end-users.

Chapter 5 dealt with the development of an architecture to address the issues of (2). We started by deriving the requirements (subsec. 5.1.3) based on the observations (subsec. 5.1.1) of the State of The Art (chapter 3) and proposed the Model-Agent Concept (MAC), which is based on the concept of software agents making the design models alive.

Different to other architectural proposals, by implementing the MAC, an application remains aware of its design model semantics and thus enables the continuous evolvement of the applications to support evolutionary prototyping allowing to change the models even at run-time. The MAC benefits from the advantages of software agents regarding its capabilities to coordinate and to act autonomously. The former enables the synchronization of information upon changes between the various models whereas the latter enables to flexibly configure the set of utilized interactive system models based on the actual requirements of the application domain.

Finally, in chapter 6 we started by presenting the observations and deriving the requirements in the same way we did for the previous chapters and ended up detailing the task-model to directly connect to the domain and service model, which allowed us to directly execute the task model at run-time and prototype the addition of further modalities by specifying modality-specific task trees that synchronize with the application task trees in subsection 6.2.1.4.

Further on, we introduced another model, which focuses specifically on the layout generation of user interfaces (section 6.2.3), which has not been addressed by actually model-based development approaches. The layout model consists of layouting statements, which are interpretations of the other design models and therefore enabling a layout definition that is consistent to all the other design models. Since these layouting statements can be even defined for certain contexts-of-use independently of a certain application support for new contexts of already existing applications is possible with our approach.

Finally, in subsection 6.2.2 we discuss our abstract user interface (AUI) model, which is directly derived and connected to the domain model. Different to other approaches that specify the AUI independently from the other design models, we select and enrich relevant domain model information (application objects) for the presentation of the user interface (to form abstract interaction objects). These AIOs are realized like a facade bound to a certain application object and are assembled to form an abstract user interface based on the actual set of relevant application objects at run-time. Compared to previous approaches our approach is more flexible as it can consider even sets of elements only known at run-time (for instance a list of all connected appliances) and supports reuse since existing AIO facades for a certain domain concept can be automatically considered by the system for generating a presentation in another application that reuses this concept.

8.3 Validation

8.3.1 External Validation

The external validation has been realized by application of the MASP method, the MAC architecture and all of the proposed models in the form of several case studies as part of different research projects together with the industry. A condensed but complete walk-through of the MASP methodology and the relevant tools to support the developer that have been implemented to complement our approach have been described in chapter 7. The main goal of these case studies was to show the feasibility of developing an interactive

application in a user-centered way rigorously following the process of the methodology.

Earlier case studies we did, in that only parts of the methodology have been applied, motivated the design by example approach and most of the tools. It figured out that without a tool-support the extensive abstractions via specifying the various models were very complex to understand and the learning curve for developers was very low.

End-user involvement in early phases of the design phase figured out to be too complex when directly working with the models. Instead the approach to derive examples increased the understanding of the developer about the actual status of the development process.

8.3.2 Internal Validation

For the internal validation of the method, the architecture, and the proposed model additions, we pick up the requirements as identified in sec 4.1.3, sec 5.1.3, and section 6.1.3 respectively and discuss the outcome regarding each requirement after the results of this thesis have been applied.

8.3.2.1 Methodological Requirements

Requirement 1: Life-Cycle Coverage – states that the proposed method should cover all phases of a software engineering process, which includes early requirements analysis, design, prototyping, implementation, deployment, maintenance and adaptation at run-time.

Discussion: The MASP methodology covers all phases of development to realize an interactive application. Requirement analysis involves eliciting the requirements based on scenario descriptions. The design phase consists of the subsequent specification of various models; each model describes a different aspect of the application, which reduces the design complexity together with the subsequent derivation of design models from abstract to more concrete ones. The method supports the exemplary instantiation of the design models in the early phases as well as the derivation of prototypes in the later phases of the development process, which reduces the complexity of testing a model-based application design. Designers are not required to think in abstractions but instead are directly confronted with parts of the final application. Most of the method is driven by transformations of one design model to the next more concrete one and ends in a Model-Agent-driven runtime-system that loads and instantiate the design models. The only two aspects that are not covered by the design models of this methods are on the one hand realizing the backend functionality (which can be developed following well known software engineering processes) and on the other hand adding aesthetical design aspects that are not covered by the design models. This includes adding pictures and user interface designs to the user interface presentation. Deployment is done by instantiating Model-Agents; each contains a design model instance and is connected to the other agents based on the model relations. Each Model-Agent can be inspected at run-time by using a debugger tool, allowing to directly modifying the instance data, like modifying the domain, task as well as the layouting model, which directly results in a modification of

the actual user interface presentation. The adaptation of the user interface for certain contexts-of-use can be specified on the task-level to support different tasks with different modalities, on the abstract user interface level to present different domain model data for different platforms, and finally on the layout model level based on the actual context-of-use of the user, which includes his actual environment and the capabilities of the platform the user is using.

Requirement 2: Automated process with support for interactive development – stating that tool support is required to be considered by the method to reduce the complexity for the developer in specifying the models and supporting how his modifications manipulate the resulting user interface.

Discussion: The MASP method includes tool support for all phases of the development process. Since it was not the goal of this work to re-invent the wheel, already existing tools have been considered, such as the Eliciting Tasks Tool in the analysis phase or various tools to support the domain modeling like the Portege or the JIAC IV Ontology-Builder. In cases where limitations prevent the utilization of already existing tools, we implemented additions concentrating on certain aspects like done by the realization of the MASP Task Tree Editor, which complements the already existing ConcurTaskTree editor. Unique additions of our approach like the way to derive the abstract user interface model from the domain model (1), the proposal for a layouting model (2), and the approach to enable exemplary instance and prototype derivation (2) are supplemented by tools we build on our own.

Requirement 3: Automation of repeating tasks – is about the prevention of repeating work in the various phases of the methodology.

Discussion: The automation of repeating tasks has been considered in the design of the tools that have been developed to support the method. Looking at the task level, the highest level of abstraction, already existing task trees can be included. Existing task trees can be re-used on the one hand on an application design level and on the other hand for describing the application control with specific modalities. With the former concept, task trees can serve as patterns to address re-occurring tasks like for instance a user login or the functionality of a typically basic command set like often realized as a “file” and an “edit” menu. Re-using tasks results in preserving all the mappings to the domain-, abstract-, and concrete user interface level for the included task tree. The latter relies on specifying task trees to describe usage patterns for the control of an application via a certain modality or platform.

When specifying the abstract user interface, already existing domain objects can be re-used and therefore the entire abstract object definitions that have been specified for these domain objects. Re-using domain objects initially ends-up re-using the presentations for these objects in the user interface, as long as no modifications have been done to the referring abstract interaction object definitions.

Finally on the layout model level, layouting statements are defined independently of a certain application, by only specifying meta-model level interpretations that are targeting

to certain contexts-of-use. These kinds of statements can be included in the layout-model to be re-used on the one hand during the design process and on the other hand during the execution of the application at run-time. During the design-time process, meta-model statements specify for instance the derivation of the containment structure from the task tree, or toggle the orientation of elements to realize an element distribution conforming to a certain screen aspect ratio. By using layouting statements to describe individual platform capabilities the layout adaptation to these platform can be defined to be re-used as well.

Requirement 4: Support for multi-path development – refers to the flexible adaptation of the method to be implemented in different forms of organization.

Discussion: There have been a lot of different aspects identified that can be considered to support methodological flexibility. Flexibility can be defined regarding the development direction like for instance by following top-down, bottom-up or even middle-out and wide-spreading approaches. Different to these approaches, the MASP method has been designed as a top-down approach supporting an abstract to detail modeling and therefore does not explicitly consider reverse-engineering or different entry-points. Instead the method supports flexibility to the application domain of the interactive application. Thus the developer is free to select those steps in the design phase that she feels most comfortable with. After the task analysis and task and domain design, which is mandatory for a user-centered development the developer is free to choose if he requires modeling an abstract user interface when the application should support various modalities or defining a layout model if context-of-use adaptations have been identified as relevant for the final application.

Requirement 5: Support for explicit testing - states to keep the iteration cycles short and to check design results against the requirements as early and often as possible.

Discussion: The development of interactive applications that should be run on several platforms and are able to adapt to several contexts-of-use requires a lot of development effort. End-users' feedback should be considered early and often to prevent tray and error trashing. Model-based development suffers from the fact, that models are set-up to serve as abstractions that support the developer keeping an overview even for the design of complex applications. But these model-based abstractions are not suitable to be discussed with end-users to identify problems and misunderstandings. The main reason for this is because of the notations and abstractions that an end-user is not familiar with.

The MASP methodology is designed with at least one testing activity in each step of the development process and enables the developer to test the state of the application design by deriving examples, prototypes and simulations. This forces the designer to re-think the models and keeps the iteration cycles short. Testing steps of the method include the consistency checks during the analysis phase (subsection 4.2.1.3), the task process simulation (subsection 4.2.2.3), the derivation of exemplary objects to check the domain model (subsection 4.2.3.3), the abstract user interface simulation (subsection

4.2.4.3), the boxed-based layout simulation (subsection 4.2.5.2), as well as finally the final user interface preview (subsection 4.2.6.4).

Requirement 6: Clear distinction of concerns – means the suitability to implement the method for development teams where different specialists are working together.

Discussion: The development process in the MASP method is happening by the subsequent definition of abstract models into more concrete ones. Since models are related and in most cases get directly derived from the more abstract ones, a specification of all the models in parallel is not supported by the method. But since each model implements specific aspects a clear distinction of concerns is supported. Beneath considering domain experts to define the domain model as well as working in close cooperation with a system analyst to identify the tasks that should be supported by the application, it is up to the designers and domain expert's to decide which data is relevant for presentation in the user interface. The layout model, as well as the derivation of the concrete model is supposed to be the work of a user interface designer, where more specifically, experts to implement specific platforms can be considered for the concrete user interface derivation as well.

8.3.2.2 Architectural Requirements

Requirement 7: Conceptual Simplicity – demands for an architecture that offers modularity to be easy understandable and extensible.

Discussion: An application following the MAC concept can be extended in two ways: First by adding another view by introducing an additional Model-Agent, for instance to support describing user interface distributions or layout adaptations or second by modifying the specification of a model to add for instance another task or another presentation for a specific platform. Whereas the former enables a flexible configuration of a set of models and corresponding Model-Agents based on the requirements of the application domain, the latter ensures that the developer always is confronted with a comprehensive view of the whole application, which reduces the possibility of introducing inconsistent manipulations.

Requirement 8: Separation of Concerns – requires the architecture to consider a model-based run-time system and clearly defines the assignment of each component.

Discussion: The MAC agent concept has been designed based on the ideas of earlier architectures and extends them by the idea of preserving the design models to drive the interactive application even at run-time. Following the MAC concept, the architecture is not modularized based on a functional decomposition like realized by the PAC-agents hierarchy or organized into loosely coupled autonomic modules like proposed in MVC, but separated into several aspects. Each aspect specifies a complete view on the application on a certain level of abstraction. Each Model-Agent is comprised of three components: the abstraction which is realized by a model, the control part which contains the mapping to the other agents, and the instance, which stores the actual data at run-time.

Requirement 9: Performance Considerations – means that all calculations that the MASP architecture is required to perform at run-time do not affect the interaction with the user.

Discussion: During our work several performance measurements have been done to ensure that those parts of the architecture that require a lot of calculations can be performed in a reasonable time and do not bother the interaction with the user. Relevant parts of the overall MASP architecture that require a lot of calculations at run-time are the multi-level transformations (1), the layout statement evaluation and constraint solving (2), and the coordination effort through messages between the Model-Agents (3). The performance of the multi-level transformation (1) has been optimized and evaluated to work efficiently for a huge amount of parallel interaction requests, such as it usually happens on web-servers, where a lot of users request web-sites in parallel. It figured out that around 100 requests can be responded by a presentation, where we can ensure a response time from max 3,5s. Calculating the layout of a graphical presentation (2) has been measured for selecting the relevant statements based on the actual context-of-use and for solving the generated constraint network (section 7.6). Since we select the relevant statements to produce the constraint network (which required <1ms) first and thereafter construct and solve the constraint network (between 8-20ms), we could efficiently reduce the constraint solving time for layouting one screen mask. For our reference implementation of the MAC concept, we use tuple spaces to synchronize the information between the agents (3) (section 5.2.2).

Requirement 10: Consideration of a methodology – refers to the relation of the method to the architecture. Stating that all parts of the architecture have to be subject to analysis and design in the methodology.

Discussion: An architecture based on the MAC concept requires at least four Model-Agents, namely a task-, a domain, a service-, and a concrete user interface Model-Agent. All four models are subject to analysis and design of the MASP methodology. After the models have been designed, they get directly deployed into a Model-Agent. Thus a run-time system requires the instantiation of at least four agents. The MASP method further supports designing a layout and an abstract user interface model, which can be instantiated as additional Model-Agents. The layout Model-Agent further requires contextual information to select relevant layouting statements which are stored in a context Model-Agent.

Requirement 11: Support for adaptation at run-time – refers to missing self-awareness of interactive systems. Possible adaptations of the user interface are defined during design-time and get implemented based on a set of pre-known contexts-of-use.

Discussion: The Model-Agents are constructed using three basic components. In the abstraction layer, the behavior and static structure of an agent is declaratively modeled and open to be accessed at run-time (section 5.2.2) This is different to other model-based approaches that concentrate on applying model-to-code transformations at design-time and therefore irreversibly loose the model-to-code bindings. By keeping the design

model specifications alive more flexible adaptations of the user interface are possible since adaptations can be addressed to the different abstraction levels (for instance by delimiting or modifying the set of tasks to be performed based on the actually used device, or changing the information amount selected by the AIOs based on the actual situation of the user. In section 5.2.4 we presented how we can adapt the layout even to platforms that have been unconsidered during design-time, which takes advantage of the fact that we keep models alive.

Requirement 12: Tool-support – states that the architecture should support connection points enabling tools on that a model-based development heavily depends on, direct model manipulations.

Discussion: After an interactive application has been implemented and is running maintenance work is required to fix issues like for instance adding new contexts-of-use that have not been considered at design-time. The tools that we have realized to support the MASP method support a direct synchronization and manipulation with already running applications. The MASP Builder (section 4.2.4.4) can load the domain and abstract user interface model as well as the current instance data of a running system that should be target to a modification. This data can then be manipulated offline by the designer and the resulting changes to the AIOs and the multi-level transformations can be transferred back to the run-time environment that serves the actual application. In the same way the Layout Model Generator supports retrieving the layout- and the context model to introduce new layout adaptations by introducing or manipulating layouting statements. Finally the MASP Debugger directly connects to a running application and enables access and manipulation options to all of the models' instance information.

8.3.2.3 Language Requirements

Requirement 13: Comprehensive Lifecycle support – demands for a model-based user interface specification that is comprehensible as it covers all levels of abstraction and all phases of the software-engineering lifecycle.

Discussion: We focused with the MASP on language extensions to already existing language models (like we did for the task and abstract model in section 6.2.1 and section 6.2.2 respectively) and to add model specifications where they are missing (section 6.2.3). Especially on the concrete user interface level a lot of promising work has been done as part of the UsiXML specification. By following the multi-level transformation approach (section 4.2.6) our approach is not bound to a certain concrete user interface model. Thus the concrete user interface models from UsiXML or any other representation like XUL, GIML, or UIML can be utilized.

Requirement 14: Run-time support – requires the language to be designed to be directly interpreted.

Discussion: To support run-time interpretation two aspects had to be considered: First, redundancies between all the different model abstractions have to be avoided. Redundancies can result in conflicting designs that will require the run-time system to decide for one interpretation or will prevent the direct execution if the run-time environment is not able to select one option. Second, mappings of individual model elements between the various levels of abstraction have to be clearly stated. We ensured the former by further restricting the task model temporal operators, clearly defining the various task types and by defining an abstract user interface model that does not contain ambiguous definitions in relation to the task model like done in the UsiXML specification. The latter has been ensured by explicitly annotating the application domain objects to the task tree (section 6.2.1), deriving the abstract user interface based on the specified domain objects (section 6.2.2), and by clearly distinguishing between application and interaction tasks that are mapped to services of the service model and the concrete user interface presentations respectively.

Requirement 15: Adherence to standards – refers to a language model that is compatible to already existing languages and standards.

Discussion: Adherence to standards has been realized by supporting on the one hand import and export of our MASP Task Tree Editor (MTTE) to support the file format of the Eliciting Tasks Tool and the ConcurTaskTree editor, which enabled us to set up a tool chain that includes these tools. On the other hand, our language models have been designed to consider already existing standards as well as to be flexibly applied to support new standards. Thus, the abstract user interface model can be utilized for several different formats. Whereas the MASP Builder utilized ontologies for the domain model, UPnP Profiles, Java classes as well as data based on class structures or database tables can be utilized for deriving the AUI model as well. Finally the service model is currently realized to support Java as well as Universal Plug and Play calls, but web services and remote procedure calls could be applied as well.

Requirement 16: Consideration of semantic data – requires considering the increasing amount of content that is structured in a way that the data’s semantics can be captured by a machine.

Discussion: In section 6.2.2 we presented the derivation of abstract interaction objects based on a domain model that is structured by using ontologies. Since the selection and enrichment activities are directly bound to the underlying domain objects structure, the processing of the abstract user interface can benefit from the domain model semantics. For instance we benefit from the aggregation, composition, and inheritance relationships and are able to automatically select the correct abstract interaction objects based on these relationships. Further on we can refer to different AIO’s for a certain domain object, depending whether it is part of an aggregation, or an inheritance relationship. The former one could for instance result in a condensed representation, whereas the latter scales the level of detail of the representation based on the inheritance level.

8.4 Summary of Contributions

The contribution of this work consists of three aspects: a method, an architecture and language model additions. Whereas the method is the main driving force and motivation for this thesis, the proposal for an architecture awaking the design-models alive at run-time as well as several language additions supplement this work.

1. A method for a model-based development of interactive applications that:
 - Is comprehensive in the term that it covers the complete software engineering process
 - Is tool-supported for each step of the development process to reduce the often mentioned complexity of model-based user interface design.
 - Follows a strict user-centered development approach, which includes to derive exemplary results and prototypes in each step of the development process (“design by example”) to test the model-based designs as often and early as possible. These prototypes can be used by a developer to retrieve feedback from the users and prevent misunderstandings and tray and error trashing.
2. An architecture for realizing model-based run-time environments based on software agents. The architecture is designed to:
 - Separate a system based on the design and implementation of aspects, each containing a comprehensive view to ensure that additions and manipulations can be applied consistently with the complete system view in mind.
 - Keep the design-time models alive in a declarative way and derives the model synchronization based on the model mappings, which enables ad-hoc context-of-use adaptations based on reasoning about the models at run-time.
 - Be easily configurable and extensible by supporting the flexible configuration and addition of new Model-Agents based on the requirements of the interactive application and the language models that should be utilized for implementation.
3. Extension and definition of further models, which enable:
 - Direct execution of the involved models to reduce the gap between design-time and run-time.
 - Introduction of explicit mappings to the other models as part of the model definition supporting the designer to think about the model-to-model relations during the design and enable the automatic derivation to the agent-based run-time environment.
 - A specification of the layout by interpreting the other design models’ information for layout generation. The layout model can be easily enhanced to support adaptations to new contexts-of-use independently from a specific application and is efficiently processable at run-time.
 - The specification of an abstract user interface that considers the domain model semantics and maintains the relation to the domain model at run-time.

8.5 Future Work

Several things still remain to be done around the method, the tools and the language models that have been proposed in this work. The vision of model-based development is to completely remove the gap between design-time and run-time, which means that all the design-time models, can be defined in a way that they all can be directly executable without the need for an implementation phase. If tools became intuitive and easy to use that even the end-user can be involved in designing an application, in the future each end user is able to manipulate and adapt applications to fulfill exactly his personal needs to be efficiently accessed on devices and modalities he is feeling most comfortable with.

Whereas realizing this vision still involves a lot of research efforts, this work enables several options to be further investigated:

User interface migration and distribution: User interface migration enables the realization of follow-me services and requires the user interface layout to be adapted to every platform that is located close to the user. Whereas the calculation and adaptation of all possible user interface migration scenarios might be done as part of the design-process, user interface distribution requires access to the model semantics at run-time. Since it is up to the user or to the environment to decide which part of the user interface should be presented by which platform or modalities and therefore all possible distributions can't be handled during design process.

Fusion and fission in multi-modal user interfaces: Fusion and Fission of multiple modalities is a complex problem. The famous “put that there” example [Bolt, 1980] demonstrates that fusion algorithms require a lot of semantics to merge inputs from the user via various modalities. In [Blumendorf et al., 2006] we already proposed an initial concept for managing the fusion and fission by benefiting from the run-time semantics and relying on a multi-level propagation of input and output events. The basic idea of this approach is to handle the fusion and fission separately for each level of abstraction. In case a fusion is not possible information are propagated to a higher level until merging is possible.

Usability testing: Since by our approach all semantics are declaratively stated and can be manipulated at run-time, wizard-of-oz testing can be easily applied. For instance the debugging tool can be used to directly manipulate a running application based on what the user inputs. Since these manipulations can be done for every model not only primitive user inputs can be entered to the system but certain tasks can be marked as done or domain concepts can be changed to reflect what the user means.

Rapid prototyping: By evolving the application at run-time and adding or changing the models prototyping can be performed efficiently in two ways. First, the application's presentation or functionality (like for instance the task flow) can be manipulated instantly to consider the user's feedback. Second, new platforms and modalities can be tested directly with a running application. Especially if the application

requires a specific setup to be run, like for instance a smart environment, prototyping can become time-consuming, if each change of the prototype requires another design, implementation and deployment cycle to be tested. An initial approach by specifying modality specific task trees that can be flexible added to an already running application has been described in subsection 6.2.1 and [Feuerstack et al., 2007] respectively.

Automatic usability evaluation: Interactive applications that are able to adapt to several contexts-of-use including platforms, modalities, users, as well as certain environment conditions can only tested to a limited extend by a traditional usability evaluation. The pure amount of possible representations of the user interface demand for approaches that can automate at least parts of the testing process. A way to evaluate the usability of a system is to compare the system interaction model with the mental model of a user. Mismatches of both models can indicate usability issues. Since when following the MAC concept, the system interaction model is already explicitly and declaratively existent in each Model-Agent, automated usability evaluation can benefit from this information. A first prototype that follows such an automated usability evaluation approach has been described in [Feuerstack et al., 2008], where we connected the MASP run-time environment to the MeMo workbench.

Bibliography

- [Abrams and Helms, 2002] Abrams, M. and Helms, J. (2002). User interface markup language (UIML) specification version 3.0. Technical report, Harmonia Inc.
- [Abrams et al., 1999] Abrams, M., Phanouriou, C., Batongbacal, A. L., Williams, S. M., and Shuster, J. E. (1999). UIML: An appliance-independent XML user interface language. *Computer Networks*, 31(11-16):1695–1708.
- [Alencar et al., 1995] Alencar, P. S. C., Cowan, D. D., de Lucena, C. J. P., and Nova, L. C. M. (1995). Formal specification of reusable interface objects. In *SSR '95: Proceedings of the 1995 Symposium on Software Reusability*, pages 88–96, New York, NY, USA. ACM Press.
- [Avgeriou and Zdun, 2005] Avgeriou, P. and Zdun, U. (2005). A pattern language for documenting software architectures. In *Proceedings of Tenth European Conference on Pattern Languages of Programs (EuroPLOP)*, Irsee, Germany.
- [Azevedo et al., 2000] Azevedo, P., Merrick, R., and Roberts, D. (2000). OVID to AUIML - user-oriented interface modelling. In *UML'2000 Workshop "Towards a UML Profile for Interaction Systems" (TUPIS'2000)*.
- [Badros et al., 2001] Badros, J., G., Borning, A., Stuckey, and J., P. (2001). The cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction*, 8(4):267–306.
- [Balbo et al., 2004] Balbo, S., Ozkan, N., and Paris, C. (2004). *The Handbook of Task Analysis for Human-Computer Interaction*, chapter "Choosing the right task modeling notation: A taxonomy", pages 445 – 466. Lawrence Erlbaum Associates (LEA).
- [Balme et al., 2004] Balme, L., Demeure, A., Barralon, N., Coutaz, J., and Calvary, G. (2004). CAMELEON-RT: A software architecture reference model for distributed, migratable, and plastic user interfaces. In Markopoulos, P., Eggen, B., Aarts, E. H. L., and Crowley, J. L., editors, *Ambient Intelligence: Second European Symposium on Ambient Intelligence (EUSAI) 2004, Eindhoven, The Netherlands, November 8-11, 2004. Proceedings*, pages 291–302. Springer, ISBN 3-540-23721-6.
- [Balzert et al., 1996] Balzert, H., Hofmann, F., Kruschinski, V., and Niemann, C. (1996). The JANUS application development environment - generating more than the user interface. In Vanderdonckt, J., editor, *Proceedings of the Second International Workshop on Computer-Aided Design of User Interfaces (CADUI)*, pages 183–208, Namur, Belgium. Presses Universitaires de Namur, ISBN 2-87037-232-9.

Bibliography

- [Barclay et al., 1999] Barclay, P. J., Griffiths, T., McKirdy, J., Paton, N. W., Cooper, R., and Kennedy, J. B. (1999). The Teallach tool: Using models for flexible user interface design. In *Proceedings of the third International Conference on Computer-aided Design of User Interfaces (CADUI)*, pages 139–158. Kluwer Academic Publishers Norwell, MA, USA, ISBN:0-7923-6078-8.
- [Bass et al., 1998] Bass, L., Clements, P., and Kazman, R. (1998). *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., ISBN: 0201199300, Boston, MA, USA.
- [Bass et al., 1992] Bass, L., Faneuf, R., Little, R., Mayer, N., Pellegrino, B., Reed, S., Seacord, R., Sheppard, S., and Szczur, M. R. (1992). A metamodel for the runtime architecture of an interactive system: The UIMS tool developers workshop. *ACM SIGCHI Bulletin*, 24(1):32–37.
- [Berti et al., 2004] Berti, S., Correani, F., Mori, G., Paternò, F., and Santoro, C. (2004). TERESA: A transformation-based environment for designing and developing multi-device interfaces. In *ACM CHI 2004, Extended Abstract*, volume II, pages 793–794, Vienna, Austria. ACM Press.
- [Berti and Paternò, 2005] Berti, S. and Paternò, F. (2005). Migratory multimodal interfaces in multidevice environments. In *ICMI '05: Proceedings of the 7th International Conference on Multimodal interfaces*, pages 92–99, New York, NY, USA. ACM Press.
- [B'far, 2004] B'far, R. (2004). *Mobile Computing Principles: Designing and Developing Mobile Applications with UML and XML*. Cambridge University Press, New York, NY, USA.
- [Blumendorf et al., 2006] Blumendorf, M., Feuerstack, S., and Albayrak, S. (2006). Event-based synchronization of model-based multimodal user interfaces. In Pleuss, A., den Bergh, J. V., Hussmann, H., Sauer, S., and Boedcher, A., editors, *Proceedings of the MoDELS'06 Workshop on Model Driven Development of Advanced User Interfaces*. CEUR-WS.org.
- [Blumendorf et al., 2007] Blumendorf, M., Feuerstack, S., and Albayrak, S. (2007). Multimodal user interaction in smart environments: Delivering distributed user interfaces. In Mühlhäuser, M., Ferscha, A., and Aitenbichler, E., editors, *Constructing Ambient Intelligence: AmI 2007 Workshops Darmstadt*. Springer-Verlag GmbH.
- [Bodard et al., 1994] Bodard, F., Hennebert, A.-M., Leheureux, J.-M., Provot, I., and Vanderdonckt, J. (1994). A model-based approach to presentation: A continuum from task analysis to prototype. In *The Eurographics Workshop on Design, Specification and Verification of Interactive Systems*, pages 25–39.
- [Bodart et al., 1995a] Bodart, F., Hennebert, A.-M., Leheureux, J.-M., Provot, I., Sacré, B., and Vanderdonckt, J. (1995a). Towards a systematic building of software architecture: The TRIDENT methodological guide. In Palanque, P. and Bastide, R., editors,

Bibliography

- Design, Specification and Verification of Interactive Systems '95*, pages 262–278, Wien. Springer-Verlag.
- [Bodart et al., 1995b] Bodart, F., Hennebert, A.-M., Leheureux, J.-M., Provot, I., Vanderdonckt, J., and Zucchinetti, G. (1995b). *Critical Issues in User Interface Systems Engineering*, chapter Chapter 7: Key activities for a development Methodology of Interactive Applications, pages pp. 109–134. Springer-Verlag.
- [Boehm, 1988] Boehm, B. W. (1988). A spiral model of software development and enhancement. *IEEE Computer*, ISSN: 0018-9162, 21(5):61–72.
- [Boese and Feuerstack, 2003] Boese, J.-H. and Feuerstack, S. (2003). Adaptive user interfaces for ubiquitous access to agent-based services. In *Agentcities ID3, Workshop on Human Agent Interaction*, Barcelona, Spain.
- [Bolt, 1980] Bolt, R. A. (1980). "Put that there": Voice and gesture at the graphics interface. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 262–270, New York, NY, USA. ACM Press.
- [Borning et al., 1987] Borning, A., Duisberg, R., Freeman-Benson, B., Kramer, A., and Woolf, M. (1987). Constraint hierarchies. In *OOPSLA '87: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 48–60, New York, NY, USA. ACM Press.
- [Boyer, 2006] Boyer, J. M. (2006). XForms 1.1. Technical report, World Wide Web Consortium, W3C Candidate Recommendation, Online at <http://www.w3.org/TR/xforms11/>, last checked 12/11/08.
- [Braudes, 1990] Braudes, R. E. (1990). *A framework for conceptual consistency verification*. PhD thesis, George Washington University, Washington, DC, USA.
- [Brown, 1997] Brown, J. (1997). HCI and requirements engineering: Exploring human-computer interaction and software engineering methodologies for the creation of interactive software. *SIGCHI Bulletin*, 29(1):32–35.
- [Brown et al., 1998] Brown, J., Graham, T. C. N., and Wright, T. (1998). The vista environment for the coevolutionary design of user interfaces. In *CHI '98: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 376–383, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.
- [Byrne et al., 1994] Byrne, M. D., Wood, D., Sukaviriya, P. N., Foley, J. D., and Kieras, D. E. (1994). Automating interface evaluation. In Plaisant, C., editor, *CHI Conference Companion*, page 216. ACM.
- [Bäumer et al., 1996] Bäumer, D., Bischofberger, W. R., Lichter, H., and Züllighoven, H. (1996). User interface prototyping: Concepts, tools, and experience. In *ICSE '96: Proceedings of the 18th International Conference on Software Engineering*, pages 532–541, Washington, DC, USA. IEEE Computer Society.

Bibliography

- [Calvary et al., 2004] Calvary, G., Coutaz, J., Dâassi, O., Balme, L., and Demeure, A. (2004). Towards a new generation of widgets for supporting software plasticity: The "comet". In Bastide, R., Palanque, P. A., and Roth, J., editors, *Proceedings of Engineering Human Computer Interaction and Interactive Systems, Joint Working Conferences EHCI-DSVIS 2004*. ISSN 0302-9743, volume 3425 of *Lecture Notes in Computer Science*, pages 306–324, Berlin/Heidelberg. Springer.
- [Calvary et al., 2001] Calvary, G., Coutaz, J., and Thevenin, D. (2001). A unifying reference framework for the development of plastic user interfaces. In *EHCI '01: Proceedings of the 8th IFIP International Conference on Engineering for Human-Computer Interaction*, pages 173–192, London, UK. Springer-Verlag.
- [Calvary et al., 2003] Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J. (2003). A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308.
- [Calvary et al., 2002] Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Souchon, N., Bouillon, L., Florins, M., and Vanderdonckt, J. (2002). Plasticity of user interfaces: A revised reference framework. In *TAMODIA '02: Proceedings of the First International Workshop on Task Models and Diagrams for User Interface Design*, pages 127–134. INFOREC Publishing House Bucharest.
- [Card et al., 1980] Card, S. K., Moran, T. P., and Newell, A. (1980). The keystroke-level model for user performance with interactive systems. *Communications of the ACM*, 23:396–410.
- [Card et al., 1983] Card, S. K., Moran, T. P., and Newell, A. (1983). *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Ass. Publ., Hillsdale. ISBN 0898592437.
- [Carroll, 1992] Carroll, J. M. (1992). Creating a design science of human-computer interaction. In *Proceedings of the International Conference on Future Tendencies in Computer Science, Control and Applied Mathematics*, pages 205–215, London, UK. Springer-Verlag.
- [Carroll, 1999] Carroll, J. M. (1999). Five reasons for scenario-based design. In *HICSS: Proceedings of the Thirty-Second Annual Hawaii International Conference on System Sciences*, ISBN:0-7695-0001-3, volume 03, page 3051, Washington, DC, USA. IEEE Computer Society.
- [Case et al., 2000] Case, R. B., Maes, S. H., and Raman, T. V. (2000). Position paper for the W3C/WAP workshop on the web device independent authoring. Technical report. Online at: <http://www.w3.org/2000/10/DIAWorkshop/statements.html>, last checked 12/11/08.
- [Chamberlain et al., 2002] Chamberlain, D., Díaz, A., Gisolfi, D., Konuru, R. B., Lucassen, J. M., MacNaught, J., Maes, S. H., Merrick, R., Mundel, D., Raman, T. V.,

Bibliography

- Ramaswamy, S., Schaeck, T., Thompson, R., and Wiecha, C. (2002). WSXL: A web services language for integrating end-user experience. In Kolski, C. and Vanderdonckt, J., editors, *CADUI'2002: Proceedings of the 4th International Conference on Computer-Aided Design of User Interfaces*, pages 35–50. Kluwer.
- [Chu et al., 2004] Chu, H.-H., Song, H., Wong, C., Kurakake, S., and Katagiri, M. (2004). Roam, a seamless application framework. *Journal of Systems and Software*, 69(3):pages 209–226.
- [Clerckx et al., 2006a] Clerckx, T., den Bergh, J. V., and Coninx, K. (2006a). Modeling multi-level context influence on the user interface. In *PERCOMW '06: Proceedings of the 4th annual IEEE international conference on Pervasive Computing and Communications Workshops*, pages 57–61, Washington, DC, USA. IEEE Computer Society.
- [Clerckx et al., 2004a] Clerckx, T., Luyten, K., and Coninx, K. (2004a). Dynamo-AID: A design process and a runtime architecture for dynamic model-based user interface development. In *The 9th IFIP Working Conference on Engineering for Human-Computer Interaction Jointly with The 11th International Workshop on Design, Specification and Verification of Interactive Systems (EHCI/DS-VIS)*, ISBN 3-540-26097-8, volume 3425, pages 77–95, Springer, Berlin.
- [Clerckx et al., 2004b] Clerckx, T., Luyten, K., and Coninx, K. (2004b). The mapping problem back and forth: Customizing dynamic models while preserving consistency. In Slavík, P. and Palanque, P. A., editors, *Proceedings of the 3rd annual conference on Task Models and Diagrams (TAMODIA)*, ISBN:1-59593-000-0, volume 86, pages 33–42, Prague, Czech Republic. ACM International Conference Proceeding Series.
- [Clerckx et al., 2007] Clerckx, T., Vandervelpen, C., and Coninx, K. (2007). Task-based design and runtime support for multimodal user interface distribution. In *Proceedings of Engineering Interactive Systems 2007 (EHCI-HCSE-DSVIS'07)*.
- [Clerckx et al., 2006b] Clerckx, T., Vandervelpen, C., Luyten, K., and Coninx, K. (2006b). A prototype-driven development process for context-aware user interfaces. In *Proceedings of the 5th International Workshop, Task Models and Diagrams for Users Interface Design (TAMODIA 2006)*, volume 4385 of *Lecture Notes in Computer Science*, pages 339–354, Berlin / Heidelberg.
- [Cockburn, 1997] Cockburn, A. A. (1997). Goals and use cases. *Journal of Object-Oriented Programming*, ISSN 0896-8438, 10(5):35–40.
- [Collignon et al., 2008] Collignon, B., Vanderdonckt, J., and Calvary, G. (2008). An intelligent editor for multi-presentation user interfaces. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC 2008)*, pages 1634–1641, New York, NY, USA.
- [Coninx et al., 2003] Coninx, K., Luyten, K., Vandervelpen, C., den Bergh, J. V., and Creemers, B. (2003). Dygimes: Dynamically generating interfaces for mobile comput-

Bibliography

- ing devices and embedded systems. In Chittaro, L., editor, *Mobile HCI*, volume 2795 of *Lecture Notes in Computer Science*, pages 256–270. Springer.
- [Coutaz, 1987a] Coutaz, J. (1987a). The construction of user interfaces and the object paradigm. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOOP '87)*, ISBN:3-540-18353-1, volume 276 of *Lecture Notes In Computer Science*, pages 135–144, London, UK. Springer-Verlag.
- [Coutaz, 1987b] Coutaz, J. (1987b). PAC: An object oriented model for implementing user interfaces. *SIGCHI Bulletin*, 19(2):37–41.
- [Coutaz, 1989] Coutaz, J. (1989). Architectural models for interactive software. In Cook, S., editor, *European Conference on Object-oriented Programming - ECOOP '89*, pages 382–399. Cambridge University Press.
- [Coutaz et al., 1995] Coutaz, J., Nigay, L., and Salber, D. (1995). Agent-based architecture modelling for interactive systems. Technical Report SM/WP53, LGI-IMAG, Grenoble.
- [Coutaz and Thevenin, 1999] Coutaz, J. and Thevenin, D. (1999). Plasticity of user interfaces: Framework and research agenda. In *Proceedings of IFIP INTERACT'99: 7th IFIP Conference on Human-Computer Interaction*, Mobile Systems, pages 110–117, Edinburgh, Scotland.
- [Coyette et al., 2004] Coyette, A., Faulkner, S., Kolp, M., Limbourg, Q., and Vanderdonckt, J. (2004). Sketchixml: Towards a multi-agent design tool for sketching user interfaces based on usixml. In Ph. Palanque, P. Slavik, M. W., editor, *Proc. of 3rd Int. Workshop on Task Models and Diagrams for user interface design TAMODIA 2004*, pages pp. 75–82, Prague. ACM Press.
- [Coyette and Limbourg, 2006] Coyette, A., V. J. and Limbourg, Q. (2006). Sketchixml: An informal design tool for user interface early prototyping. In M. Risoldi, V. A., editor, *Proceedings of RISE 2006 Workshop on Rapid User Interface Prototyping Infrastructures Applied to Control Systems RUIPICAS 2006*, Geneva.
- [Davis and Bersoff, 1991] Davis, A. M. and Bersoff, E. H. (1991). Impacts of life cycle models on software configuration management. *Communications of the ACM*, 34(8):104–118.
- [de Baar et al., 1992] de Baar, D. J. M. J., Foley, J. D., and Mullet, K. E. (1992). Coupling application design and user interface design. In *CHI '92: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 259–266, New York, NY, USA. ACM Press.
- [Demeure et al., 2006] Demeure, A., Calvary, G., Coutaz, J., and Vanderdonckt, J. (2006). The Comets Inspector: Manipulating multiple user interface representations simultaneously. In *Proceedings of 6th International Conference on Computer-Aided Design of User Interfaces (CADUI 2006)*, pages 167–174, Berlin. Springer-Verlag.

Bibliography

- [den Bergh, 2006] den Bergh, J. V. (2006). *High-Level User Interface Models for Model-Driven Design of Context-Sensitive User Interfaces*. PhD thesis, Hasselt University and transnational University of Limburg School of Information Technology.
- [den Bergh and Coninx, 2004a] den Bergh, J. V. and Coninx, K. (2004a). Contextual ConcurTaskTrees: Integrating dynamic contexts in task based design. In *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications, Workshops*, pages 13–17, Los Alamitos, CA, USA. IEEE Computer Society.
- [den Bergh and Coninx, 2004b] den Bergh, J. V. and Coninx, K. (2004b). Model-based design of context-sensitive interactive applications: A discussion of notations. In *TA-MODIA '04: Proceedings of the 3rd annual conference on Task models and diagrams*, pages 43–50, New York, NY, USA. ACM Press.
- [Dittmar and Forbrig, 2004] Dittmar, A. and Forbrig, P. (2004). The influence of improved task models on dialogues. In Jacob, R. J. K., Limbourg, Q., and Vanderdonckt, J., editors, *Proceedings of Fourth International Conference on Computer-Aided Design of User Interfaces (CADUI 2004)*, pages 1–14, Funchal, Portugal. Kluwer, ISBN 1-4020-3145-9.
- [D'Souza and Wills, 1999] D'Souza, D. F. and Wills, A. C. (1999). *Objects, components, and frameworks with UML: The catalysis approach*. Addison-Wesley Longman Publishing Co., Inc., ISBN: 0201310120, Boston, MA, USA.
- [Dubinko, 2003] Dubinko, M. (2003). *XForms Essentials*. O'Reilly & Associates, Inc., ISBN: 0596003692, Sebastopol, CA, USA.
- [Duke et al., 1994] Duke, D., Faconti, G., Harrison, M., and Paternó, F. (1994). Unifying views of interactors. In *AVI '94: Proceedings of the Workshop on Advanced Visual Interfaces*, pages 143–152, New York, NY, USA. ACM, ISBN:0-89791-733-2.
- [Duke and Harrison, 1993] Duke, D. J. and Harrison, M. D. (1993). Abstract interaction objects. *Computer Graphics Forum*, 12(3):25–36.
- [Duke and Harrison, 1995] Duke, D. J. and Harrison, M. D. (1995). Event model of human-system interaction. *IEE/BCS Software Engineering Journal*, 10(1):3–12.
- [Eisenstein et al., 2001] Eisenstein, J., Vanderdonckt, J., and Puerta, A. R. (2001). Applying model-based techniques to the development of UIs for mobile computers. In *Intelligent User Interfaces*, pages 69–76.
- [Feuerstack et al., 2007] Feuerstack, S., Blumendorf, M., and Albayrak, S. (2007). Prototyping of multimodal interactions for smart environments based on task models. In *Constructing Ambient Intelligence: AmI 2007 Workshops Darmstadt*.
- [Feuerstack et al., 2008] Feuerstack, S., Blumendorf, M., Kern, M., Kruppa, M., Quade, M., Runge, M., and Albayrak, S. (2008). Automated usability evaluation during model-based interactive system development. In *Proceedings of the 7th International workshop on TAsk MOdels and DIAGrams*, Pisa, Italy.

Bibliography

- [Feuerstack et al., 2006] Feuerstack, S., Blumendorf, M., Lehmann, G., and Albayrak, S. (2006). Seamless home services. In Maña, A. and Lotz, V., editors, *Developing Ambient Intelligence*. Springer. Proceedings of the First International Conference on Ambient Intelligence Developments (AmID'06).
- [Florins, 2006] Florins, M. (2006). *Graceful Degradation: a Method for Designing Multiplatform Graphical User Interfaces*. PhD thesis, Université Catholique de Louvain, Louvain-la-Neuve, Belgium.
- [Florins et al., 2006a] Florins, M., Simarro, F. M., Vanderdonckt, J., and Michotte, B. (2006a). Splitting rules for graceful degradation of user interfaces. In *AVI '06: Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 59–66, New York, NY, USA. ACM Press.
- [Florins et al., 2006b] Florins, M., Simarro, F. M., Vanderdonckt, J., and Michotte, B. (2006b). Splitting rules for graceful degradation of user interfaces. In *IUI '06: Proceedings of the 11th international conference on Intelligent user interfaces*, pages 264–266, New York, NY, USA. ACM Press.
- [Floyd, 1984] Floyd, C. (1984). *Approaches to Prototyping*, chapter "A Systematic Look at Prototyping", pages 1–17. Springer Verlag.
- [Fogarty and Hudson, 2004] Fogarty, J. and Hudson, S. (2004). Gadget: A toolkit for optimization-based approaches to interface and display generation. In *ACM Transactions on Graphics, Special Issue: Proceedings of the 2004 SIGGRAPH Conference*, volume 23, pages 730–730. ACM, ISSN:0730-0301.
- [Foley and Kovacevic, 1988] Foley, J.D., G. C. K. W. and Kovacevic, S. (1988). A knowledge-based user interface management system. In *Human Factors in Computing Systems CHI 88 Proceedings*.
- [Foley et al., 1988] Foley, J., Chul, W., Kovacevic, S., and Murray, K. (1988). The user interface design environment. *SIGCHI Bull.*, 20(1):77–78.
- [Foley et al., 1990] Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F. (1990). *Computer Graphics: Principles and Practice*. The Systems Programming Series. Addison-Wesley, pub-AW:adr, second edition.
- [Forbrig et al., 2004] Forbrig, P., Dittmar, A., Reichart, D., and Sinnig, D. (2004). From models to interactive systems tool support and ximl. In *Proceedings of the First International Workshop on Making Model-based User Interface Design Practical: Usable and Open Methods and Tools*.
- [Gajos and Weld, 2004] Gajos, K. and Weld, D. S. (2004). Supple: Automatically generating user interfaces. In *IUI '04: Proceedings of the 9th International Conference on Intelligent User Interfaces*, pages 93–100, New York, NY, USA. ACM Press.

Bibliography

- [Gajos and Weld, 2005] Gajos, K. and Weld, D. S. (2005). Preference elicitation for interface optimization. In *UIST '05: Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology*, pages 173–182, New York, NY, USA. ACM Press.
- [Gamma et al., 2000] Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. (2000). *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Amsterdam.
- [Gennari et al., 2002] Gennari, J., Musen, M. A., Fergerson, R. W., Grosso, W. E., Crubezy, M., Eriksson, H., Noy, N. F., and Tu, S. W. (2002). The evolution of protégé: An environment for knowledge-based systems development. In *Proceedings of Knowledge-Based Systems Development*.
- [Ghezzi et al., 1991] Ghezzi, C., Jazayeri, M., and Mandrioli, D. (1991). *Fundamentals of software engineering*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- [Gram and Cockton, 1997] Gram, C. and Cockton, G., editors (1997). *Design principles for interactive software*. Springer, London, UK. ISBN: 0412724707.
- [Gray et al., 1998] Gray, P., Cooper, R., Kennedy, J., Barclay, P., and Griffiths, T. (1998). A lightweight presentation model for database user interfaces. In *Proceedings of the 4th ERCIM Workshop on 'User Interfaces for All'*, number 16 in Position Papers: Information Filtering and Presentation, page 14. ERCIM.
- [Gray et al., 1992] Gray, W. D., John, B. E., and Atwood, M. E. (1992). Project ernestine: Validating a GOMS analysis for predicting and explaining real-world task performance. *Human-Computer Interaction*, 8(3):pp. 2237–309.
- [Green, 1985] Green, M. (1985). Report on dialogue specification tools. In Pfaff, G. E., editor, *User Interface Management Systems, Proceedings of the workshop on User Interface Management Systems*, pages pp. 9–20, Berlin. FRG. Springer Verlag.
- [Griffiths et al., 1999] Griffiths, T., Barclay, P. J., McKirdy, J., Paton, N. W., Gray, P. D., Kennedy, J. B., Cooper, R., Goble, C. A., West, A., and Smyth, M. (1999). Teallach: A model-based user interface development environment for object databases. In *User Interfaces to Data Intensive Systems*, pages 86–96.
- [Griffiths et al., 1998a] Griffiths, T., McKirdy, J., Forrester, G., Paton, N. W., Kennedy, J. B., Barclay, P. J., Cooper, R., Goble, C. A., and Gray, P. D. (1998a). Exploiting model-based techniques for user interfaces to databases. In *Proceedings of the IFIP TC2/WG 2.6 Fourth Working Conference on Visual Database Systems 4*, pages 21–46. ISBN:0-412-84400-1.
- [Griffiths et al., 1998b] Griffiths, T., McKirdy, J., Paton, N. W., Kennedy, J. B., Cooper, R., Barclay, P. J., Goble, C. A., Gray, P. D., Smyth, M., West, A., and Dinn, A. (1998b). An open-model-based interface development system: The teallach approach.

Bibliography

- In Markopoulos, P. and Johnson, P., editors, *Supplementary Proceedings of 5th International Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS '98)*, volume 2, pages 34–50. Eurographics Association.
- [Hartmann and Sure, 2004] Hartmann, J. and Sure, Y. (2004). An infrastructure for scalable, reliable semantic portals. *Intelligent Systems, IEEE*, 19(3):58–65.
- [Hartson and Gray, 1992] Hartson, H. R. and Gray, P. D. (1992). Temporal aspects of tasks in the user action notation. *Human Computer Interaction*, 7:pp.1–45.
- [Hewett et al., 1992] Hewett, T. T., Baecker, R., Card, S., Carey, T., Gasen, J., Mantei, M., Perlman, G., Strong, G., and Verplank, W. (1992). ACM SIGCHI curricula for human-computer interaction. Technical report, ACM SIGCHI, New York, NY, USA. Chairman-Thomas T. Hewett.
- [Hill, 1992] Hill, R. D. (1992). The abstraction-link-view paradigm: Using constraints to connect user interfaces to applications. In *CHI '92: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 335–342, New York, NY, USA. ACM Press.
- [Hirsch et al., 2008] Hirsch, B., Fricke, S., Kroll-Peters, O., and Konnerth, T. (2008). Agent programming in practice - experiences with the JIAC IV agent framework. In *Sixth International Workshop "From Agent Theory to Agent Implementation" held at the Seventh International Conference on Autonomous Agents and Multiagent Systems*.
- [Hix et al., 1993] Hix, D., Hartson, and Rex, H. (1993). *Developing User Interfaces: Ensuring Usability Through Product and Process*. John Wiley & Sons, New York, New York. ISBN: 0471578134.
- [Hosobe, 2001] Hosobe, H. (2001). A modular geometric constraint solver for user interface applications. In *UIST '01: Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology*, pages 91–100, New York, NY, USA. ACM Press.
- [Hussey and Carrington, 1996] Hussey, A. and Carrington, D. (1996). Comparing two user-interface architectures: MVC and PAC. Technical report, Technical Report No. 95-33, Software Verification Research Centre, Dept. of Computer Science, University of Queensland, Australia.
- [IEEE Computer Society, 1990] IEEE Computer Society (1990). IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990, The Institute of Electrical and Electronics Engineers, Inc, Piscataway, NJ 08854 USA.
- [Isoda, 2003] Isoda, S. (2003). A critique of UML's definition of the use-case class. In Stevens, P., Whittle, J., and Booch, G., editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 280–294. Springer.

Bibliography

- [Jacobson, 1992] Jacobson, I. (1992). *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Massachusetts.
- [Johnson, 1992] Johnson, P. (1992). *People and Computers VIII*, chapter The ADEPT user interface design environment. Cambridge University Press.
- [Johnson et al., 1995] Johnson, P., Johnson, H., and Wilson, S. (1995). *Rapid Prototyping of User Interfaces Driven by Task Models*, pages 209 – 246. John Wiley & Sons, New York, NY, USA.
- [Johnson et al., 1993] Johnson, P., Wilson, S., Markopoulos, P., and Pycck, J. (1993). Adept: Advanced design environment for prototyping with task models. In *CHI '93: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, page 56, New York, NY, USA. ACM Press.
- [Johnson, 1991] Johnson, H. & Johnson, P. (1991). Task knowledge structures: Psychological basis and integration into system design. In *Acta Psychologica*, volume 78, pages 3–26.
- [Kaltz et al., 2005] Kaltz, J. W., Lohmann, S., and Ziegler, J. (2005). Eine komponentenorientierte Architektur für die kontext-sensitive Adaption von Web-Anwendungen. In Cremers, A. B., Manthey, R., Martini, P., and Steinhage, V., editors, *GI Jahrestagung (2)*, volume 68 of *LNI*, pages 129–133. GI.
- [Katsurada et al., 2003] Katsurada, K., Nakamura, Y., Yamada, H., and Nitta, T. (2003). XISL: a language for describing multimodal interaction scenarios. In *ICMI '03: Proceedings of the 5th International Conference on Multimodal Interfaces*, pages 281–284, New York, NY, USA. ACM Press.
- [Klug and Kangasharju, 2005] Klug, T. and Kangasharju, J. (2005). Executable task models. In *Proceedings of TAMODIA 2005*, pages 119–122, Gdansk, Poland. ACM Press.
- [Kost, 2006] Kost, S. (2006). *Dynamically Generated Multi-modal Application Interfaces*. PhD thesis, Technical University Dresden, Faculty Computer Science, Leipzig.
- [Laurel, 1990] Laurel, B. (1990). *The Art of Human-Computer Interface Design*. Addison-Wesley Professional, Boston, MA, USA. ISBN: 0201517973.
- [Lim and Long, 1994] Lim, K. Y. and Long, J. (1994). *The Muse Method for Usability Engineering*. Cambridge University Press, New York, NY, USA.
- [Limbourg, 2004] Limbourg, Q. (2004). *Multi-Path Development of User Interfaces*. PhD thesis, Université Catholique de Louvain, Institut d'Administration et de Gestion (IAG), Louvain-la-Neuve, Belgium.
- [Limbourg and Vanderdonckt, 2004a] Limbourg, Q. and Vanderdonckt, J. (2004a). Addressing the mapping problem in user interface design with USIXML. In *TAMODIA*

Bibliography

- '04: *Proceedings of the 3rd Annual Conference on Task Models and Diagrams*, pages 155–163, New York, NY, USA. ACM Press.
- [Limbourg and Vanderdonckt, 2004b] Limbourg, Q. and Vanderdonckt, J. (2004b). USIXML: A user interface description language supporting multiple levels of independence. In Matera, M. and Comai, S., editors, *ICWE Workshops*, pages 325–338. Rinton Press. ISBN 1-58949-046-0.
- [Limbourg et al., 2004a] Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., Florins, M., and Trevisan, D. (2004a). USIXML: A user interface description language for context-sensitive user interfaces. In *Proceedings of the ACM AVI'2004 Workshop "Developing User Interfaces with XML: Advances on User Interface Description Languages"*, pages 55–62.
- [Limbourg et al., 2004b] Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., and López-Jaquero, V. (2004b). USIXML: A language supporting multi-path development of user interfaces. In Bastide, R., Palanque, P. A., and Roth, J., editors, *EHCI/DS-VIS*, volume 3425 of *Lecture Notes in Computer Science*, pages 200–220. Springer.
- [Limbourg et al., 2000] Limbourg, Q., Vanderdonckt, J., and Souchon, N. (2000). The task-dialog and task-presentation mapping problem: Some preliminary results. In *Interactive Systems Design, Specification, and Verification*, volume 1946/2001, pages 227–246, Berlin / Heidelberg. Springer.
- [Linton et al., 1989] Linton, M. A., Vlissides, J. M., and Calder, P. R. (1989). Composing user interfaces with interviews. *IEEE Computer*, 22(2):8–22.
- [Lu et al., 1999] Lu, S., Paris, C., and Linden, K. V. (1999). Toward the automatic construction of task models from object-oriented diagrams. In *Proceedings of the IFIP TC2/TC13 WG2.7/WG13.4 Seventh Working Conference on Engineering for Human-Computer Interaction*, pages 169–189, Deventer, The Netherlands. Kluwer, B.V.
- [Luo, 1994] Luo, P. (1994). A human-computer collaboration paradigm for bridging design conceptualization and implementation. In Paternó, F., editor, *Design, Specification and Verification of Interactive Systems '94*, pages 129–147, Heidelberg. Springer-Verlag.
- [Luyten, 2004] Luyten, K. (2004). *Dynamic User Interface Generation for Mobile and Embedded Systems with Model-Based User Interface Development*. PhD thesis, Transnationale Universiteit Limburg, School voor Informatietechnologie.
- [Luyten et al., 2003a] Luyten, K., Clerckx, T., Coninx, K., and Vanderdonckt, J. (2003a). Derivation of a dialog model from a task model by activity chain extraction. In *Interactive Systems: Design, Specification, and Verification, 10th International Workshop (DSV-IS)*, pages 203–217, Funchal, Madeira Island, Portugal.

Bibliography

- [Luyten and Coninx, 2001] Luyten, K. and Coninx, K. (2001). An XML-based runtime user interface description language for mobile computing devices. In Johnson, C., editor, *DSV-IS*, volume 2220 of *Lecture Notes in Computer Science*, pages 1–15. Springer.
- [Luyten et al., 2003b] Luyten, K., Creemers, B., and Coninx, K. (2003b). Multi-device layout management for mobile computing devices. Technical Report TR-LUC-EDM-0301, Limburgs Univeristair Centrum – Expertise Centre for Digital Media.
- [Luyten et al., 2003c] Luyten, K., Laerhoven, T. V., Coninx, K., and Reeth, F. V. (2003c). Runtime transformations for modal independent user interface migration. *Interacting with Computers*, 15(3):329–347.
- [Luyten et al., 2006a] Luyten, K., Thys, K., Vermeulen, J., and Coninx, K. (2006a). A generic approach for multi-device user interface rendering with UIML. In *CADUI 2006: Computer-Aided Design of User Interfaces V*.
- [Luyten et al., 2002] Luyten, K., Vandervelpen, C., and Coninx, K. (2002). Migratable user interface descriptions in component-based development. In *DSV-IS*, volume 2545/2002 of *Lecture Notes in Computer Science*, pages 44–58, Berlin / Heidelberg. Springer. ISBN: 978-3-540-00266-6.
- [Luyten et al., 2006b] Luyten, K., Vermeulen, J., and Coninx, K. (2006b). Constraint adaptability of multidevice user interfaces. In Richter, K., Nichols, J., Gajos, K., and Seffah, A., editors, *Proceedings of the CHI*06 Workshop on The Many Faces of Consistency in Cross Platform Design*.
- [Mackinlay, 1986] Mackinlay, J. (1986). Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics (TOG)*, 5(2):110–141.
- [Martijn van Welie, 2003] Martijn van Welie, Gerrit C. van der Veer Vrije Universiteit, F. o. S. D. o. C. S. (2003). Pattern languages in interaction design: Structure and organization. In *Proceedings of Interact 2003*.
- [Martikainen et al., 2002] Martikainen, A., Paakki, J., and Kähkipuro, P. (2002). An xml-based framework for developing usable and reusable user interfaces for multi-channel environments. University of Helsinki.
- [Menkhaus, 2002] Menkhaus, G. (2002). *Adaptive User Interface Generation in a Mobile Computing Environment*. PhD thesis, Department of Computer Science, University of Salzburg, Salzburg, Austria.
- [Michotte, 2008] Michotte, B., V. J. (2008). Grafixml, a multi-target user interface builder based on usixml. In *Proceedings of 4th International Conference on Autonomic and Autonomous Systems ICAS 2008*. IEEE Computer Society Press.
- [Montero et al., 2005] Montero, F., López-Jaquero, V., Vanderdonckt, J., González, P., Lozano, M. D., and Limbourg, Q. (2005). Solving the mapping problem in user interface design by seamless integration in idealxml. In *Proceedings of 12th Int. Workshop*

Bibliography

- on Design, Specification, and Verification of Interactive Systems (DSV-IS'2005)*, pages 161–172.
- [Montero and López-Jaquero, 2006] Montero, F. S. and López-Jaquero, V. (2006). Fast hi-fi prototyping by using idealxml. Technical report, Departamento de Sistemas Informáticos, Universidad de Castilla-La Mancha.
- [Mori et al., 2002] Mori, G., Paternò, F., and Santoro, C. (2002). Ctte: Support for developing and analyzing task models for interactive system design. *IEEE Trans. Software Eng.*, 28(8):797–813.
- [Mori et al., 2003] Mori, G., Paternò, F., and Santoro, C. (2003). Tool support for designing nomadic applications. In *IUI '03: Proceedings of the 8th International Conference on Intelligent User Interfaces*, pages 141–148, New York, NY, USA. ACM Press.
- [Mori et al., 2004] Mori, G., Paternò, F., and Santoro, C. (2004). Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE Transactions on Software Engineering*, 30(8):507–520.
- [Mueller et al., 2001] Mueller, A., Forbrig, P., and Cap, C. H. (2001). Model-based user interface design using markup concepts. In Johnson, C., editor, *Proceedings of the 8th International Workshop on Interactive Systems: Design, Specification, and Verification - Revised Papers*, volume 2220 of *Lecture Notes in Computer Science*, pages 16–27. Springer. ISBN:3-540-42807-0.
- [Muller and Kuhn, 1993] Muller, M. J. and Kuhn, S. (1993). Participatory design. *Commun. ACM*, 36(6):24–28.
- [Myers et al., 2000] Myers, B., Hudson, S. E., and Pausch, R. (2000). Past, present, and future of user interface software tools. *ACM Transactions on Human-Computer Interaction*, 7(1):3–28.
- [Myers, 1989] Myers, B. A. (1989). User-interface tools: Introduction and survey. *IEEE Softw.*, 6(1):15–23.
- [Myers, 1995] Myers, B. A. (1995). User interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 2(1):64–103.
- [Myers et al., 1997] Myers, B. A., McDaniel, R. G., Miller, R. C., Ferrenzy, A. S., Faulring, A., Kyle, B. D., Mickish, A., Klimovitski, A., and Doane, P. (1997). The amulet environment: New models for effective user interface software development. *IEEE Trans. Softw. Eng.*, 23(6):347–365.
- [Nichols et al., 2002] Nichols, J., Myers, B. A., Higgins, M., Hughes, J., Harris, T. K., Rosenfeld, R., and Pignol, M. (2002). Generating remote control interfaces for complex appliances. In *UIST'02: Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology*, pages 161–170, New York, NY, USA. ACM Press.

Bibliography

- [Nigay and Coutaz, 1991] Nigay, L. and Coutaz, J. (1991). Building user interfaces: Organizing software agents. Technical report, ESPRIT'91, Project Nr. 3066: AMODEUS (Assimilating Models of DEsigners, Users and Systems).
- [Nigay and Coutaz, 1993] Nigay, L. and Coutaz, J. (1993). A design space for multimodal systems: concurrent processing and data fusion. In *CHI '93: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 172–178, New York, NY, USA. ACM Press.
- [Nigay and Coutaz, 1997a] Nigay, L. and Coutaz, J. (1997a). *Formal Methods in Human Computer Interaction*, chapter "Software Architecture Modelling: Bridging two Worlds using Ergonomics and Software Properties", pages 49–73. Springer Verlag.
- [Nigay and Coutaz, 1997b] Nigay, L. and Coutaz, J. (1997b). Multifeature systems: The care properties and their impact on software design. In *Intelligence and Multimodality in Multimedia Interfaces*.
- [Noy et al., 2001] Noy, N. F., Sintek, M., Decker, S., Crubézy, M., Fergerson, R. W., and Musen, M. A. (2001). Creating semantic web contents with. In *Protégé-2000. IEEE Intelligent Systems (2001)*, pages 60–71.
- [Olsen, 1986] Olsen, D. R. (1986). Mike: the menu interaction kontrol environment. *ACM Trans. Graph.*, 5(4):318–344.
- [Paternò and Faconti, 1993] Paternò, F. and Faconti, G. (1993). On the use of lotos to describe graphical interaction. In *HCI'92: Proceedings of the Conference on People and Computers VII*, pages 155–173, New York, NY, USA. Cambridge University Press.
- [Paternò, 1999] Paternò, F. (1999). *Model-Based Design and Evaluation of Interactive Applications*. Applied Computing. Springer-Verlag, Berlin. ISBN: 185233155.
- [Paternò, 2005] Paternò, F. (2005). Model-based tools for pervasive usability. *Interacting with Computers*, 17(3):291–315.
- [Paternò et al., 1998] Paternò, F., Breedvelt-Schouten, I. M., and de Koning, N. (1998). Deriving presentations from task models. In Chatty, S. and Dewan, P., editors, *Proceedings of the IFIP TC2/TC13 WG2.7/WG13.4 Seventh Working Conference on Engineering for Human-Computer Interaction*, volume 150 of *IFIP Conference Proceedings*, pages 319–337. Kluwer.
- [Paternò and Mancini, 1999] Paternò, F. and Mancini, C. (1999). Developing task models from informal scenarios. In *Proceedings of ACM CHI 99 Conference on Human Factors in Computing Systems*, volume 2 of *Late-breaking results: Exploring the frontiers of interface design*, pages 228–229.
- [Paternò et al., 1997] Paternò, F., Mancini, C., and Meniconi, S. (1997). Concurtask-trees: A diagrammatic notation for specifying task models. In Howard, Steve, H.-J. L. G., editor, *Proceedings of Interact'97*, Part of the INTERACT: IFIP International Conference On Human-Computer Interaction conference series. Chapman and Hall.

Bibliography

- [Phanouriou, 2000] Phanouriou, C. (2000). *UIML: A Device-Independent User Interface Markup Language*. PhD thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia.
- [Plomp and Mayora-Ibarra, 2002] Plomp, J. and Mayora-Ibarra, O. (2002). A generic widget vocabulary for the generation of graphical and speech-driven user interfaces. *International Journal of Speech Technology*, 5(1):39–47. ISSN: 1381-2416.
- [Potts, 1995] Potts, C. (1995). Using schematic scenarios to understand user needs. In *DIS '95: Proceedings of the Conference on Designing Interactive Systems*, pages 247–256, New York, NY, USA. ACM Press.
- [Pribeanu, 2002] Pribeanu, C. (2002). Investigating the relation between the task model and the domain model in a task-based approach to user interface design. In *TAMODIA '02: Proceedings of the First International Workshop on Task Models and Diagrams for User Interface Design*, pages 78–85. INFOREC Publishing House Bucharest.
- [Pribeanu, 2007] Pribeanu, C. (2007). Tool support for handling mapping rules from domain to task models. In *Task Models and Diagrams for Users Interface Design*, 4385/2007, pages 16–23. Springer.
- [Puerta, 2001] Puerta, A. (2001). XIML: A common representation for interaction data. In *Proceedings of ACM conference on Intelligent User Interfaces (IUI'01)*, pages 214–215. ACM.
- [Puerta and Eisenstein, 2001] Puerta, A. and Eisenstein, J. (2001). Ximl: A universal language for user interfaces. Technical report, RedWhale Software, 227 Town & Country, Palo Alto.
- [Puerta and Eisenstein, 2003] Puerta, A. and Eisenstein, J. (2003). *Multiple User Interfaces: Engineering and Application Framework*, chapter "Developing a Multiple User Interface Representation Framework for Industry". Wiley and Sons. ISBN: 9780470854440.
- [Puerta, 1996] Puerta, A. R. (1996). The MECANO project: Comprehensive and integrated support for model-based interface development. In *Computer-Aided Design of User Interfaces (CADUI'96)*, pages 19–36.
- [Puerta, 1997] Puerta, A. R. (1997). A model-based interface development environment. *IEEE Softw.*, 14(4):40–47.
- [Puerta et al., 1999] Puerta, A. R., Cheng, E., Ou, T., and Min, J. (1999). MOBILE: User-centered interface building. In *CHI '99: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 426–433, New York, NY, USA. ACM Press.
- [Puerta and Eisenstein, 1999] Puerta, A. R. and Eisenstein, J. (1999). Towards a general computational framework for model-based interface development systems. In *IUI99*:

Bibliography

- International Conference on Intelligent User Interfaces*, pages 171–178, New York, NY, USA. ACM. ISBN:1-58113-098-8.
- [Reichard et al., 2004] Reichard, D., Forbrig, P., and Dittmar, A. (2004). Task models as basis for requirements engineering and software execution. In *Proceedings of TAMODIA 2004*, pages 51 – 57, Prague, Czeck Republic. ACM Press.
- [Richter, 2006] Richter, K. (2006). Transformational consistency. In Calvary, G., editor, *CADUI'2006 Computer-AIDED Design of User Interfaces*, Berlin / Heidelberg. Springer. ISBN: 1402058209.
- [Rieger et al., 2005] Rieger, A., Cisse, R., Feuerstack, S., Wohltorf, J., and Albayrak, S. (2005). An agent-based architecture for ubiquitous multimodal user interfaces. In *International Conference on Active Media Technology*, Takamatsu, Kagawa, Japan.
- [Rodríguez and Scapin, 1997] Rodríguez, F. G. and Scapin, D. L. (1997). Editing MAD* task descriptions for specifying user interfaces, at both semantic and presentation levels. In Harrison, M. D. and Torres, J. C., editors, *Design, Specification and Verification of Interactive Systems '97*, Eurographics, pages 193–208, Wien. Springer-Verlag. Proceedings of the Eurographics Workshop in Granada, Spain, June 4 – 6, 1997.
- [Roth and Mattis, 1990] Roth, S. F. and Mattis, J. (1990). Data characterization for intelligent graphics presentation. In *CHI '90: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 193–200, New York, NY, USA. ACM Press.
- [Scacchi, 2002] Scacchi, W. (2002). *Encyclopedia of Software Engineering, 2nd. Edition*, chapter "Process Models in Software Engineering". New York, NY, USA.
- [Schaefer and Bleul, 2006] Schaefer, R. and Bleul, S. (2006). Towards object oriented, uiml-based interface descriptions for mobile devices. In *6th International Conference on Computer-Aided Design of User Interfaces CADUI 2006*, Bucharest, Romania.
- [Schaefer et al., 2004] Schaefer, R., Bleul, S., and Mueller, W. (2004). A novel dialog model for the design of multimodal user interfaces. In Bastide, R., Palanque, P. A., and Roth, J., editors, *EHCI/DS-VIS*, volume 3425 of *Lecture Notes in Computer Science*, pages 221–223. Springer.
- [Schlungbaum, 1997] Schlungbaum, E. (1997). Individual user interfaces and model-based user interface software tools. In *IUI '97: Proceedings of the 2nd International Conference on Intelligent User Interfaces*, pages 229–232, New York, NY, USA. ACM Press.
- [Schäfer, 2007a] Schäfer, R. (2007a). *Model-based Developent of Multimodal and Multi-Device User Interfaces in Context-Aware Environments*. PhD thesis, C-LAB Publication, Band 25, Shaker Verlag, Aachen.

Bibliography

- [Schäfer, 2007b] Schäfer, R. (2007b). SerCHo Deliverable D511.2: Ergebnisse der Usabilityanalyse Showroom. Technical report, WIK-Consult, Bad-Honef, Germany.
- [Sears, 1995] Sears, A. (1995). AIDE: A step toward metric-based interface development tools. In *UIST '95 - Proceedings of the 8th annual ACM Symposium on User Interface Software and Technology*, pages 101–110, New York, NY, USA. ACM Press.
- [Sessler and Albayrak, 2001] Sessler, R. and Albayrak, S. (2001). Service-ware framework for developing 3g mobile services. In *The Sixteenth International Symposium on Computer and Information Sciences, ICSIS XVI*.
- [Shan, 1990] Shan, Y.-P. (1990). MoDE: A UIMS for smalltalk. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, pages 258–268, New York, NY, USA. ACM. ISBN:0-201-52430-X.
- [Shevlin and Neelamkavil, 1991] Shevlin, F. and Neelamkavil, F. (1991). Designing the next generation of UIMSs. In *Proceedings of the Workshop on User Interface Management Systems and Environments on User Interface Management and Design*, pages 123–133, New York, NY, USA. Springer-Verlag New York, Inc. ISBN:0-387-53454-7.
- [Shneiderman and Maes, 1997] Shneiderman, B. and Maes, P. (1997). Direct manipulation vs. interface agents. *interactions*, 4(6):42–61.
- [Shneiderman and Plaisant, 2004] Shneiderman, B. and Plaisant, C. (2004). *Designing the User Interface: Strategies for Effective Human-Computer Interaction (4th Edition)*. Pearson Addison Wesley, Boston, MA, USA.
- [Singh and Green, 1989] Singh, G. and Green, M. (1989). A high-level user interface management system. In *CHI '89: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 133–138, New York, NY, USA. ACM Press.
- [Sluys, 2004] Sluys, M. V. (2004). Developpement d'un maquetteur d'interfaces a'l aide de microsoft visio. Master's thesis, Universite catholique de Louvain, Louvain-la-Neuve, Belgium.
- [Souchon and Vanderdonckt, 2003] Souchon, N. and Vanderdonckt, J. (2003). A review of xml-compliant user interface description languages. In Jorge, J. A., Nunes, N. J., and e Cunha, J. F., editors, *DSV-IS*, volume 2844 of *Lecture Notes in Computer Science*, pages 377–391. Springer.
- [Spriestersbach et al., 2003] Spriestersbach, A., Ziegert, T., anf Gabriel Dermmler, G. G., and Wasmund, M. (2003). Flexible pagination and layouting for device independent authoring. Technical report, AT&T Labs - Research.
- [Stanciulescu et al., 2005] Stanciulescu, A., Limbourg, Q., Vanderdonckt, J., Michotte, B., and Montero, F. (2005). A transformational approach for multimodal web user interfaces based on usixml. In *ICMI '05: Proceedings of the 7th International Conference on Multimodal Interfaces*, pages 259–266, New York, NY, USA. ACM Press.

Bibliography

- [Stirewalt, 1999] Stirewalt, R. E. K. (1999). MDL: A language for binding user-interface models. In Vanderdonckt, J. and Puerta, A. R., editors, *CADUI*, pages 159–170. Kluwer.
- [Stirewalt and Rugaber, 2000] Stirewalt, R. E. K. and Rugaber, S. (2000). The model-composition problem in user-interface generation. *Automated Software Engineering*, 7:101–124.
- [Sukaviriya and Foley, 1990] Sukaviriya, P. and Foley, J. D. (1990). Coupling a UI framework with automatic generation of context-sensitive animated help. In *UIST '90: Proceedings of the 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 152–166, New York, NY, USA. ACM Press.
- [Sukaviriya, 1988] Sukaviriya, P. N. (1988). Dynamic construction of animated help from application context. In *ACM Symposium on User Interface Software and Technology*, pages 190–202.
- [Sumner et al., 1997] Sumner, T., Bonnardel, N., and Kallak, B. H. (1997). The cognitive ergonomics of knowledge-based design support systems. In *CHI '97: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 83–90, New York, NY, USA. ACM Press.
- [Szekely et al., 1992] Szekely, P., Luo, P., and Neches, R. (1992). Facilitating the exploration of interface design alternatives: The HUMANOID model of interface design. In *CHI '92: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 507–515, New York, NY, USA. ACM Press.
- [Szekely et al., 1993] Szekely, P., Luo, P., and Neches, R. (1993). Beyond interface builders: Model-based interface tools. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems*, Model-Based UI Development Systems, pages 383–390.
- [Szekely, 1996] Szekely, P. A. (1996). Retrospective and challenges for model-based interface development. In Bodart, F. and Vanderdonckt, J., editors, *DSV-IS 96: 3rd International Eurographics Workshop on Design, Specification, and Verification of Interactive Systems*, pages 1–27. Springer.
- [Szekely et al., 1995] Szekely, P. A., Sukaviriya, P. N., Castells, P., Muthukumarasamy, J., and Salcher, E. (1995). Declarative interface models for user interface construction tools: The MASTERMIND approach. In *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*, pages 120 – 150. ISBN:0-412-72180-5.
- [Tam et al., 1998] Tam, R. C.-M., Maulsby, D., and Puerta, A. R. (1998). Utel: A tool for eliciting user task models from domain experts. In *Proceedings of the 3rd International Conference on Intelligent User Interfaces (IUI'98)*, pages 77–80, San Francisco, California, United States. ISBN:0-89791-955-6.

Bibliography

- [Teorey et al., 1986] Teorey, T. J., Yang, D., and Fry, J. P. (1986). A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys (CSUR)*, 18(2):197–222. ISSN:0360-0300.
- [Tidwell, 2005] Tidwell, J. (2005). *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly Media, Inc.
- [Totterdell and Boyle, 1990] Totterdell, P. and Boyle, E. (1990). *Adaptive User Interfaces*, chapter "The Evaluation of Adaptive Systems", pages pp. 161–194. Academic Press, North-Holland.
- [Tuguldur et al., 2008] Tuguldur, E.-O., Hefler, A., Hirsch, B., and Albayrak, S. (2008). Toolipse: An IDE for development of JIAC applications. In *Proceedings of PRO-MAS08: Programming Multi-Agent Systems*.
- [Vanderdonckt, 2005] Vanderdonckt, J. (2005). A MDA-compliant environment for developing user interfaces of information systems. In Pastor, O. and e Cunha, J. F., editors, *CAiSE 2005: Advanced Information Systems Engineering.*, volume 3520 of *Lecture Notes in Computer Science*, pages 16–31. Springer.
- [Vanderdonckt and Berquin, 1999] Vanderdonckt, J. and Berquin, P. (1999). Towards a very large model-based approach for user interface development. In *Proceedings of the 1999 User Interfaces to Data Intensive Systems*, pages 76–85, Washington, DC, USA. IEEE Computer Society. ISBN:0-7695-0262-8.
- [Vanderdonckt and Bodart, 1993] Vanderdonckt, J. and Bodart, F. (1993). Encapsulating knowledge for intelligent automatic interaction objects selection. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, pages 424–429, New York, NY, USA. ACM Press. ISBN:0-89791-575-5.
- [Vanderdonckt et al., 2000] Vanderdonckt, J., Limbourg, Q., Eisenstein, J., and Puerta, A. (2000). Using a task model to derive presentation and dialog structure. Technical report, RedWhale Software, San Mateo, CA, USA.
- [Vanderdonckt et al., 2003] Vanderdonckt, J., Limbourg, Q., and Florins, M. (2003). Deriving the navigational structure of a user interface. In *Proceedings of IFIP INTERACT'03: Human-Computer Interaction*, 2: HCI methods, page 455.
- [Veer et al., 1996] Veer, G. C. V. D., Lenting, B. F., and Bergevoet, B. A. J. (1996). GTA: Groupware task analysis - modeling complexity. *Acta Psychologica*, 91:297–322.
- [Wegner, 1997] Wegner, P. (1997). Why interaction is more powerful than algorithms. *Communication of the ACM (CACM)*, 40(5):80–91.
- [Wilson and Johnson, 1995] Wilson, S. and Johnson, P. (1995). Empowering users in a task-based approach to design. In *DIS '95: Proceedings of the Conference on Designing Interactive Systems*, pages 25–31, New York, NY, USA. ACM Press.

Bibliography

- [Wilson and Johnson, 1996] Wilson, S. and Johnson, P. (1996). Bridging the generation gap: From work tasks to user interface designs. In Vanderdonckt, J., editor, *CADUI*, pages 77–94. Presses Universitaires de Namur.
- [Wilson et al., 1993] Wilson, S., Johnson, P., and Markopoulos, P. (1993). Beyond hacking: A model based design approach to user interface design. In *People and Computers VIII, Proceedings of the HCI'93 Conference*. Cambridge University Press.
- [Ziegert et al., 2004] Ziegert, T., Lauff, M., and Heuser, L. (2004). Device independent web applications - the author once - display everywhere approach. In *ICWE*, pages 244–255.
- [Zimmermann et al., 2002a] Zimmermann, G., Vanderheiden, G., and Gilman, A. (2002a). Universal remote console prototyping of an emerging XML based alternate user interface access standard. In *Proceedings of the Eleventh International WWW Conference*.
- [Zimmermann et al., 2002b] Zimmermann, G., Vanderheiden, G. C., and Gilman, A. S. (2002b). Universal remote console - prototyping for the alternate interface access standard. In Carbonell, N. and Stephanidis, C., editors, *User Interfaces for All*, volume 2615 of *Lecture Notes in Computer Science*, pages 524–531. Springer.

Annex

Screenshots of the Final User Interfaces of the Cooking Assistant

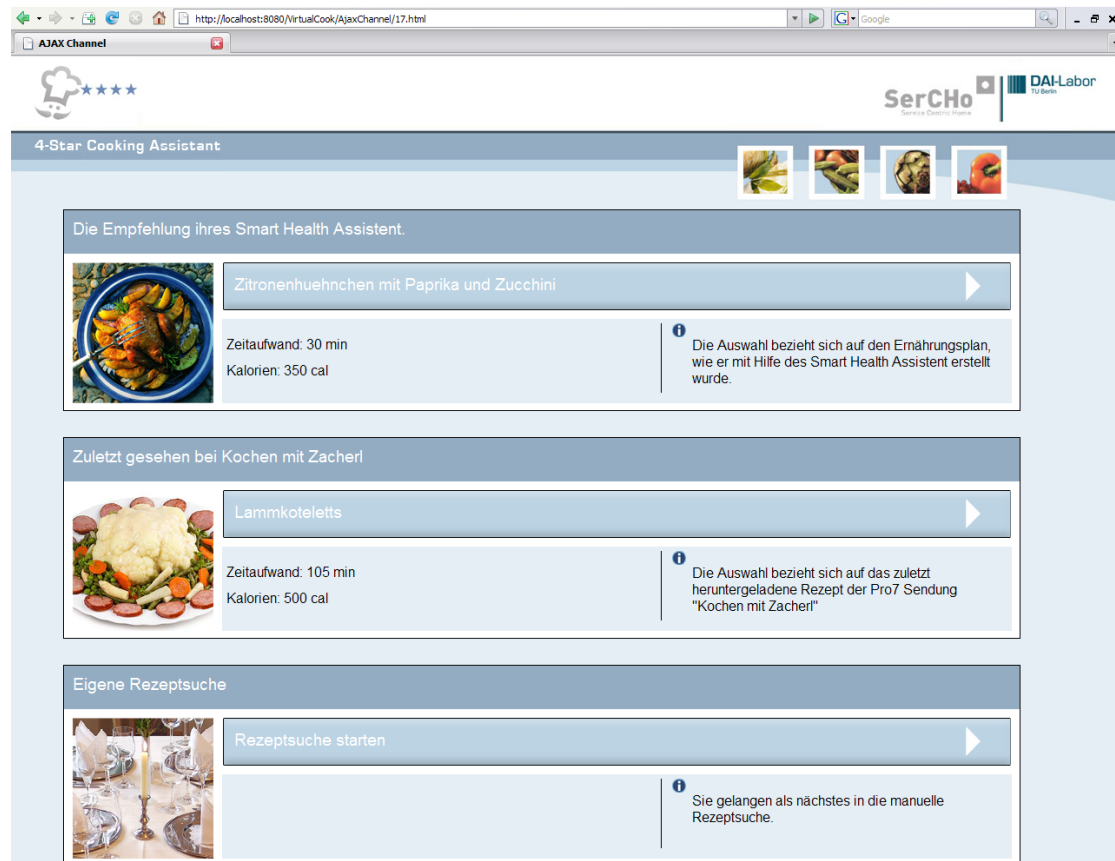


Figure 8.1: Welcome screen of the cooking assistant application. The user can choose either to cook the recipe that he has watched in the cooking show, a recipe recommendation of his health assistant, or search for a recipe in his recipe database.

Suchkriterien

In diesem Feld können Sie mit genauen Angaben zu Ihrem Gericht-Wunsch die Suche nach Ihrem Rezeptvorschlag eingrenzen.

Welche Menüart möchten Sie kochen?

☐ Hauptgericht ☐ Gebäck

☐ Nachtisch ☐ Vorspeise

Welche nationale Küche wählen Sie?

☐ Französisch ☐ Deutsch

☐ Italienisch ☐ Chinesisch

Wollen Sie gesundheitsbewusst kochen?

☐ Diätküche ☐ Fitnessküche ☐ Nein

Suche starten

Rezeptdetails

Hier werden Ihre Rezeptvorschläge mit den Details angezeigt und Sie können bestimmen, für wieviele Personen das Rezept berechnet werden soll.

Lammkoteletts

Panna Cotta
Sauer-Scharf-Suppe
Snack
Vollkornbrot mit Ei und Frischkaese
Weisskrautauflauf
Wiesenkraut Salat mit Saiblingskräpfen
Zitronenhühnchen mit Paprika und Zucchini

Lammkoteletts

Lammkoteletts mit Knödeln und Gemüse

Zeitaufwand 105 min
Kalorien 500 cal

Kochassistent starten

Figure 8.2: Screenshot of the recipe finder graphical user interface. The user can search for recipes by specifying several search criterias (left column). The right column displays the filtered recipe list and the actually selected recipe details (left-bottom).

The screenshot displays the '4-Star Cooking Assistant' web application. The interface is divided into three main sections: 'Personen', 'Einkaufsliste', and 'Zutaten Details'. The 'Personen' section on the left asks the user to enter the number of people, with the value '3' currently displayed and '+' and '-' buttons for adjustment. The 'Einkaufsliste' section in the center shows a list of ingredients with their quantities: 6.0 Knoblauchzehen, 12.0 Knödel, 150.0 g Kräuterbutter, 24.0 Lammkoteletts, 3.0 Bund Majoran (highlighted), 18.0 tl Olivenöl, 6.0 Paprika, 9.0 tl Pfeffer, 3.0 Prise Salz, and 6.0 Tomaten. The 'Zutaten Details' section on the right prompts the user to specify the quantity of the selected ingredient, with a 'Fertig' button and '+' and '-' buttons. At the bottom, there are buttons for 'Liste versenden' and 'Koch starten'. The top of the page features a '4-Star' rating icon and the 'SerCho' logo, with 'DAI-Labor' and 'TU Berlin' logos on the right. The browser's status bar at the bottom shows 'Lokales Intranet' and a 100% zoom level.

Figure 8.3: Screenshot of the shopping list generation. After the user has chosen a recipe to cook, she is asked to enter the number of persons to calculate the required ingredients amounts, which can be manipulated by the user using the right column.

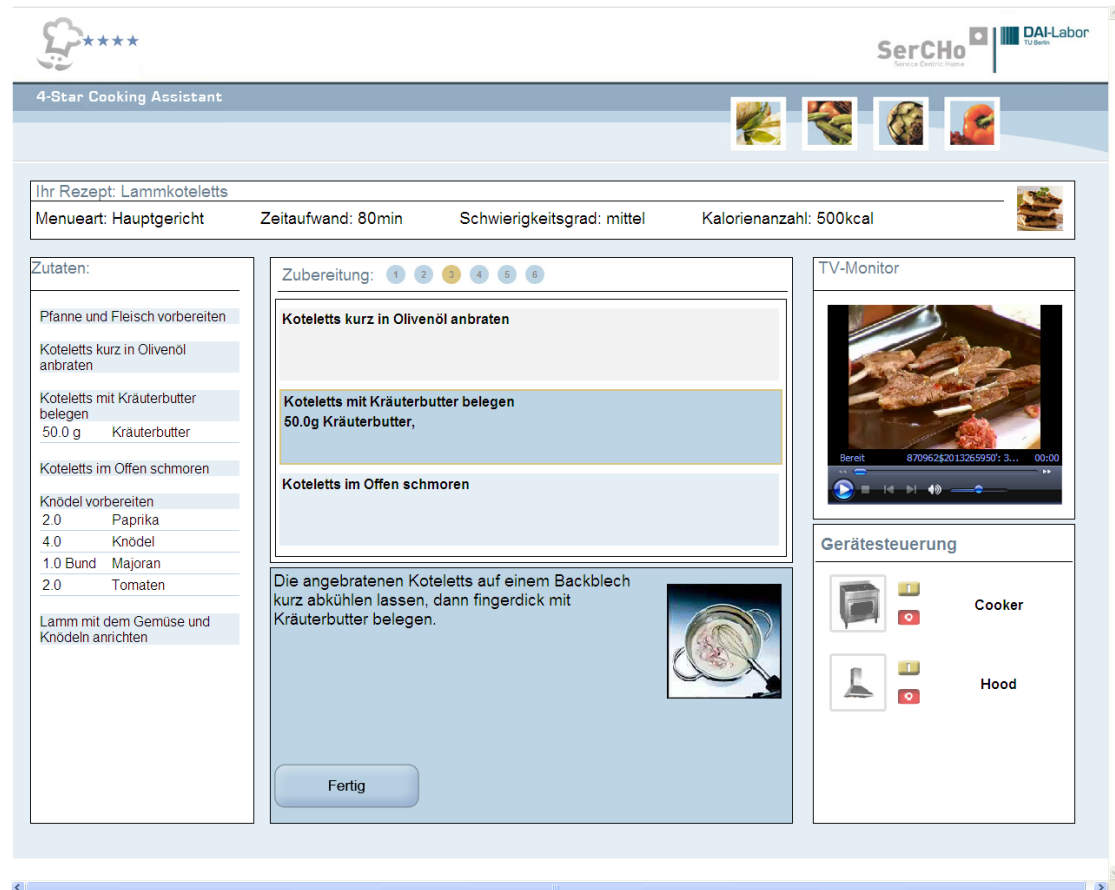


Figure 8.4: Screenshot of the cooking aid user interface that supports the user during the cooking process. The layout is inspired by a typical recipe book, but is enhanced by interactive and multimedial features. Therefore the cooking aid separates a recipe into a list of recipe steps that contain a detailed description (at the bottom of the screenshot) what the user has to do. In the left column of the screen the required ingredients are highlighted based on the actual active step and the related part of the cooking show can be reviewed in a video window at the top-right part of the screen. Finally, kitchen appliances that are required for a certain step can be controlled at the bottom-right part of the screen.



Copyright Next Generation Media 2007

Figure 8.5: The cooking assistant has been installed in the kitchen of the ambient living testbed, a modern four room apartment at the Technische Universität Berlin. The user is able to control the cooking assistant by using a touch screen that is embedded into the kitchen cubicle. All appliances that are relevant during the cooking process (like the oven and the chimney) can be controlled by the cooking assistant.

The complete Task Model of the Cooking Assistant

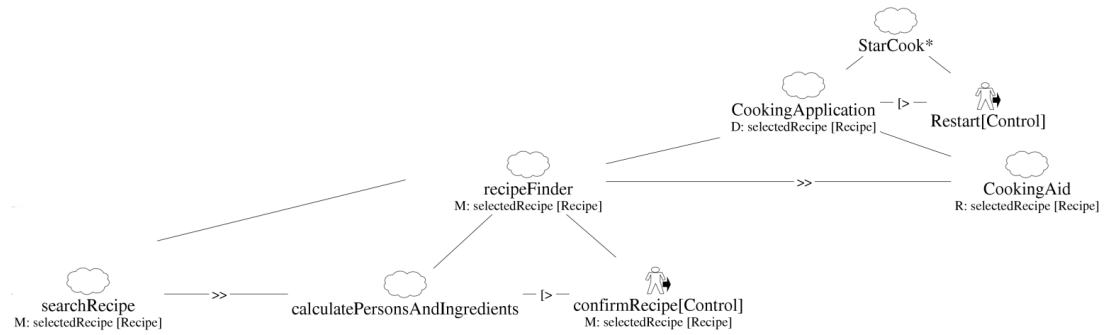


Figure 8.6: The abstract basic task tree of the cooking assistant. The task is detailed by the following figures.

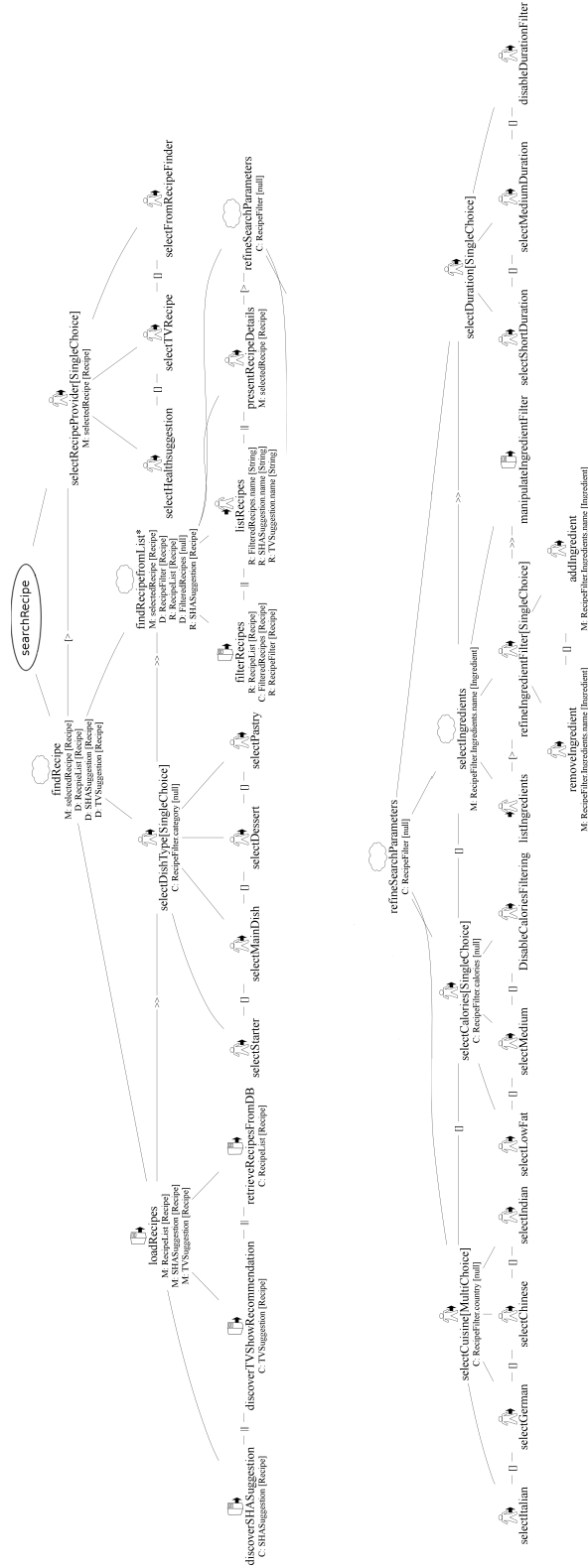


Figure 8.7: The recipe finder task tree part of the cooking assistant application.

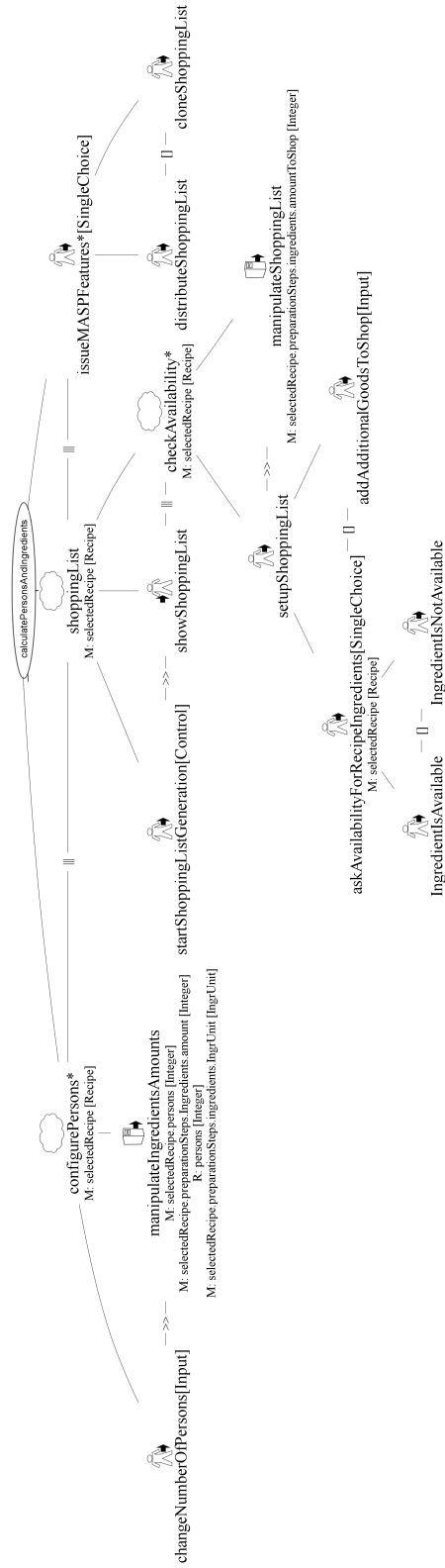


Figure 8.8: The part of the task tree that describes the calculation of the amount of ingredients.



Figure 8.9: The part of the task tree that describes the cooking aid of the cooking assistant application.

