An XML-based Framework for Developing Usable and Reusable User Interfaces for Multi-channel Applications

Report - 2002 -

Helsinki, 3rd May 2002

Pro Gradu Thesis

Department of Computer Science

UNIVERSITY OF HELSINKI

An XML-based Framework for Developing Usable and Reusable User Interfaces for Multi-channel Environments

Antti Martikainen

Pro Gradu thesis, Report - 2002 -

Department of Computer Science

UNIVERSITY OF HELSINKI

Helsinki, 3rd May 2002, 91 pages

Supervisors:

PhD Jukka Paakki, University of Helsinki

PhD Pekka Kähkipuro, SysOpen Plc

Lic.Sc. (Tech.) Marko Nieminen, Helsinki University of Technology

Abstract

User interface languages are struggling to meet the requirements for supporting universal usability in multi-channel environments. Significant problems relate to achieving a systematic user interface modelling process that is flexible enough to satisfy device specific usability requirements. The difficulties commonly result in laborious application maintenance and large development costs, or in bad usability. This thesis introduces a subset of RDIXML, an XML-based device independent user interface language. RDIXML is a task-based language that addresses device specific requirements for task sequences. While device specific functionality is allowed, the language effectively reuses the device independent task elements. A framework is sketched to interact with devices and to dynamically transform RDIXML descriptions to device specific user interfaces. An example application is used to simulate the framework's functionality.

Computing Reviews Categories: D.2.13, H.5.2, K.6.3

Keywords: User interfaces, User-centered design, User interface management systems (UIMS), Reuse models, Software maintenance

Additional keywords: Task modelling, Universal usability, Device-based navigation, Automatic user interface generation, Multi-channel architectures, XML, WWW

Contents

List of figures

List of tables

1 Introduction

Increasing number of mobile devices has led to a significant growth in the number of variety of contexts that user interface software must support [Eis00]. As new standards evolve and some devices deviate from them, the equipment used to browse the Internet varies more than in the past. In addition, new user groups with diverse educational and cultural backgrounds are discovering the Web [Shn01]. This heterogeneity altogether forms a huge design challenge and serving the users of the World Wide Web is getting more and more complicated.

These days enterprises typically deliver content to one or more channels in the World Wide Web. Existing separate channels, such as Web and voice, are typically not integrated [Lun01]. As enterprises are striving to support universal usability, they must provide multi-channel content, i.e. their Web-based services should be available for various user groups with various browsing equipment. Even though application server technologies do exist to enable faster and more robust Web application development, delivering multi-channel content is difficult. Many companies have trouble with their Web development and are in need for design methods, formalisms, languages and tools for enhancing the existing technology, even more so in the future [Bon99].

Maintaining separate software processes for separate deliverables will prove cost prohibitive over time [Lun01]. It is certainly not beneficial to describe each user interface separately for various devices, such as Web browsers, WAP phones and a choice of PDA devices. It is desirable to describe bi-directional interactions between the client and application server in a device independent manner. These abstract interactions can then be converted to meet device specific requirements by using systematic modelling principles.

Providing services to multiple types of user agents requires UI systems that concentrate on tasks, rather than on exploiting the capabilities of some specific UI environment [Ban00]. In the age of voice-based devices and variable visual UI environments, effective adaptation to various devices cannot be based on estimations on the visual properties of the devices used, as that does not support the goals of pervasive computing. Little information exists of the requirements for a task-based user interface framework's functionality. The possible implications need scrutiny especially in terms of usability of such frameworks from the developer's point of view, the amount of source code used for the UI description, run-time UI management, and usability of the resulting applications.

Today's standards do not provide a comprehensive solution that enables long-term investments for distributing content for many channels in a consistent and systematic way. Integrating small-scale technologies that only partly cover the problem field, such as CC/PP [W3C00a], requires additional effort and expertise. Today, enterprises must create application development mechanisms of their own based on various smaller technologies. Changes in small technology standards cause incompatibilities and maintaining these kinds of solutions is often tedious and difficult.

A user interface model (UIM) is a declarative specification of a user interface (UI), including its appearance, the connections between its elements or how it interacts with the underlying application functionality. It represents all the relevant aspects of a user interface in some type of interface modelling language [Pue99a]. UIM systems are generally task-oriented and use formality and higher abstraction levels to achieve device independence and UI description reuse. Abstract UI descriptions are dynamically transformed into device specific target format. However, immature mechanisms in combining the abstract model elements to concrete ones have made even most successful models applicable to only very narrow target specific areas [Pue99a]. In addition, UIM systems are typically not based on standards, which makes their adoption difficult.

XML is a text-based, structural mechanism, which can be used for partially describing the functionality of the computer programs that process them [W3C00b]. It is also well supported by industry and techniques for transforming XML-documents to other textual formats are mature. XML is thus concerned to be well suited for describing user interfaces for multi-channel environments [Mül01a]. A DTD (Document Type

Description) is used to compose a grammar that the ones implementing XML-based user interfaces must follow.

A subset of an XML-based device independent user interface language, RDIXML (Reusable Device independent Interaction XML), is introduced in this thesis. The language strives for usability and comprehensive device independence by founding the UI development process on users tasks and noticing the device dependent requirements on this matter. The task-oriented nature of the language requires a task-oriented framework to implement the language; this work should provide an insight to the functionality of the RDIXML framework and its feasibility with existing technology. Today's application development technologies typically do not support usability objectives. Hence, a major objective for the framework is to provide a usability-oriented UIM system in a package that is acceptable for today's enterprises seeking for systematic application development models. A credible multi-channel application development environment must combine qualities of efficiency, usability, pervasiveness and ease of maintenance. The design of the RDIXML framework should notice these requirements.

The scope of this thesis covers implementing the RDIXML task model grammar in DTD format and providing an overall picture of related language models. A general level design for the RDIXML framework is presented. Based on presented design solutions and the simulation, the language and the framework are analysed to reveal the benefits and drawbacks of achieved solutions; the analysis emphasizes the significance of task-oriented UI development model concerning usability, as well as the overall feasibility of this kind of task-based multi-channelled UI framework. The functionality of the framework is simulated with a use case. No usability evaluation is done for the resulted user interfaces, yet the usability power of the framework is generally discussed.

The rest of the work is organised as follows: Usability issues in multi-channel environments are discussed in chapter 2. Chapter 3 discusses user interfaces in terms of languages and UIM systems. The core of the RDIXML language is presented in chapter 4. The RDIXML framework is presented in chapter 5, where existing knowledge of multi-channel architectures is also briefly discussed. A use case to simulate the functionality of the framework is presented in chapter 6. Chapter 7 provides an analysis of the work and discusses the used methods. Finally, chapter 8 concludes the thesis by providing a summary and requirements for future work.

2 Usability Foundation

This chapter discusses some usability research areas that have significantly motivated this thesis. First, some arguments for task-based design are made. Secondly, the main difficulties in supporting universal usability are presented. Finally, requirements for providing different kinds of navigational structures in multi-channel environments are discussed.

2.1 Task-based Design

The practice of designing products in terms of the needs of their users is called user-centered design [Nor86]. Producing usable user interfaces, i.e. doing user-centered design, requires a thorough understanding of the underlying goals of the users. The UI design should be done with the goals in mind, which essentially means that the UI developers should possess the knowledge of how they are effectively achieved. Within the HCI (Human-Computer Interaction) community, task analysis is considered to make an important contribution to the design of interactive applications [Sch98]. This is due to the fact that it fundamentally is about designing user interfaces by first communicating the knowledge of user's tasks between domain experts and UI developers.

Describing user interfaces has traditionally started by describing the static visual interfaces with certain structure and controls, often called widgets [Van93, Sti98]. However, it has widely been suggested that this is a faulty starting point. Designers must think in terms of *functionality* [Bir98]. Starting the design by specifically thinking about the users task and especially the steps (subtasks) that the user must take in order to get the task completed is considered to better support the idea of user-centered design [Nor86]. If the flow of the task and its subtasks is clear, it is then easier to choose the right layout and widgets to give the user concrete tools to complete the task [Bir98]. Providing highly task specific interactive applications that allow people to focus on the actual task domain, rather than having to map that domain to the domain of computation, is the underlying idea of task-based user interface design [Sch98].

Typically, task analysis as a method is only used in the beginning of the design process and the usage of its output is unclear in the actual UI implementation process. The output is usually presented with natural or pseudo languages and thus cannot be included in the computerized UI implementation process. As UIs are built with tools that are based on screen and element structures, rather than user's tasks, it is very easy to create user interfaces that are seemingly acceptable, but do not in fact possess the flow of the original task description. When using traditional kinds of tools and techniques, there remains a gap to be filled by the builder of the actual interface; the separately described task flow offers support often too vague for a successful implementation of the actual user interface.

Use cases have commonly been used for describing the requirements to satisfy the needs of the user. Yet, the definition for a use case (Jacobson) is commonly considered vague. Cockburn has found more than 18 definitions for a use case; this strongly suggests that no true consensus about use cases exist [Coc97]. This ambiguity may also be one of the reasons for the success of use cases; the fact that the definition can be interpreted in many ways satisfies different kinds of people [For99]. Partly due to this imprecise definition and the confusion of the intended purpose of the use case concept, many use cases intermingle analysis and design, business rules and design objectives, internals and interface descriptions, combined with unessential remarks [Con01]. Hence, use cases as the output of the task analysis do not necessarily capture the essence of user's task flow. A stronger means to describe the users needs, and only them, is needed.

Prototyping is an iterative method through which the user interface is achieved by mending the faults of previous designs. A satisfactory level of usability is usually reached after a number of usability tests and redesigns. But, isn't the user interface design in the form of prototyping really a part of the solution description rather than the problem definition? Isn't there a way to achieve a better design right at the beginning? A common response is that usability cannot be captured by general specifications, and that usability is essentially implemented case-by-case. Initially, this seems acceptable. However, formal and task-based UI languages, capturing the user's case-by-case task model, have the potential for providing the best of both worlds. Being formal, they provide a direct basis for a computerized UI implementation process and, being task-based, they directly aim for producing usable user interfaces.

The Rational Unified Process (RUP) is a widespread application development model by Rational Software Corporation [Rat02]. RUP provides a development model for large-scale applications by defining best practices for the guidance of team development activities. The RUP advices development teams to use prototyping and use cases as key UI development methods. Hence, it does not seem to promote best practices for effective development of usable user interfaces for multi-channelled applications.

Hackos and Redish present a development model where the output of task analysis is used as the basis for textual scenarios, which present the task flow in its execution environment [Hac98, p. 346]. This description is then used to proceed towards the actual UI. Scenarios freely describe the user's natural movements between screens and as well describe the relevant elements in the screen during an execution of a task Based on the verbally expressed scenarios, presentation and interaction elements can be constructed to implement the user's task sequences. Even though scenarios are clearly a useful method when proceeding from abstract tasks towards concrete UIs, there remains a gap to be filled by the UI designer. Today's UI languages do not provide task-based concepts that could be used as the starting point for the development process. Implementing the given task flow in practice is still difficult.

A precise modelling mechanism would help the designer to maintain the task flow in building concrete user interfaces. The full exploitation of task analysis requires its implementation to a formal task model, i.e. there should exist a systematic transition from task identification to user interface construction [Bom98]. Importantly, it would seem that a formal task model as the basis for the UI development could reduce the number of redesigns required when using prototyping as the primary development method. Task formalisms described by the designer could lead to a more effective UI design process directly aimed at producing usable user interfaces.

2.2 Universal Usability

Universal usability is a young discipline, which aims to provide universal access for existing Internet services and data for as wide range of users as possible. According to Shneider, when 90 percent of potential users are successful users of a service, the criteria for universal usability can be seen as fulfilled [Shn00]. As new technical devices for browsing the Internet are introduced regularly, it is important for the enterprises to develop software that can be used from various devices. The diversity of user's cultural backgrounds and differences in skills and knowledge set further requirements: a user interface must adapt to the needs of different kinds of users. Mainly two fields of interest, disability access and mobile computing, have been the driving forces behind the recent interest in universal usability [Van00].

Universal usability can be divided into three branches of research [Shn00]:
Technological variance - Aims to provide access to Web-based services for all, regardless of the browsing device used.
Diversity of users - Strives for enabling use of services for all potential users not considering their knowledge and skills, cultural background, gender, disabilities or age.
Gaps in user knowledge - Bridging the gap between what users know and what they need to know.

This thesis is motivated by the technological variance of Internet devices and therefore the research areas concerning diversity of users and gaps in users knowledge are not further examined in this thesis. The emphasis of the work is on the variance in technological devices and especially on the requirements that the variance sets for producing user interfaces. The rest of the chapter presents some of the main problems in supporting the technological variance of the Internet.

Screen Size and Connection Speed

Perhaps the most significant factor in multi-channel content delivery is the variance of display resolution, which can vary from wall-sized flat screen to a small screen of a cellular phone [Eis00]. Examples of display resolutions are presented in table 1.

Device	Display resolution
Computers	1024x768
Hand-held devices	256x364
Cellular phones	48x48

Table 1: Variance of display resolution between various Internet devices

The exact resolutions of course vary according to specific manufacturers and models; the above table however gives an overall picture of the variance. To have similar content adjusted to fit the various sizes, while still keeping the user interface usable, requires substantial effort. The size of the content should fit the window as well as possible; forcing the user to scroll in search for content, especially in horizontal direction, is considered bad usability [Coo95]. The small size of the screen usually also affects the use of pictures. Pictures may not have to be altogether abandoned, but dynamic changes in the size of pictures may be a requirement. In addition, the use of frames and multiple columns may not be feasible [Nok01b]. A slow connection speed also severely restricts the size of the content, i.e. use of pictures result in slow rendering of the user interface. The use of multi-media also becomes questionable and may have to be abandoned altogether.

Various User Interface Environments

Various UI-environments run user interfaces based on different kinds of user interface languages. Idiosyncratic implementations of language standards present further

challenges. In addition, UI-environments present various restrictions on the functionality of the user interfaces, as some are not able to present pictures, for example.
All this richness causes severe problems in trying to adapt the actual content to the requirements of each device's UI-environment. Table 2 presents some examples of existing user interface languages used by Internet devices.

Device type	Language
WWW browsers	HyperText Markup Language (HTML) [W3C99a]
WAP browsers	Wireless Markup Language (WML) [WAP98]
Voice-based browsers	Voice Extensible Markup Language (VoiceXML) [Voi00]
Future WWW browsers	Extensible HyperText Markup Language (XHTML) [W3C00d]

Table 2 Various Internet devices and the UI languages they use.

The table above does not provide a comprehensive list, as that would not be meaningful in the scope of this thesis. It does however list UI languages that differ from each other considerably. For example, users of a voice-based browser do not see any UI elements and therefore cannot be expected to handle very complex navigational structures as they may prove to be overly difficult considering people's cognitive abilities [W3C99b]. The functional and presentational capabilities of the languages differ so much that content must be notably adapted to achieve usable UI solutions.

2.3 Context-sensitive Navigation

Supporting the technological variance of universal usability presents many challenges, of which not all seem to be very clearly presented in the literature concerning universal usability or task-based design. When dealing with limited display sizes and UI functionality, achieving effective business activity may require optimizations in the navigation model [Jus01, Sch01]. In terms of task-based design: the required sub-tasks in order to complete a task may vary depending on the device used. In addition, a rich UI environment with a big display promotes providing the user more than one route to complete a task, whereas in a more restricted environment, additional content could distract the user. For the more limited devices, quick and simple access to content is what counts, mainly due to low navigability and, in some cases, high connection prices and low connection speeds.

In figure 1, a regular Web browser has a four-phase sequence of actions for the completion of the task A. After completing the first action, the user may choose from three alternative sequences of actions to complete task A. The user of a PDA device with limited functionality and screen space is offered a single path for the execution of the task. The user has exactly one possible sequence of actions, through which the task can be completed. Since the action sequence only has three phases, it can be assumed that the output of the task is somewhat more limited, compared to that of the Web browser, while the task can however be completed.

It seems that in multi-channel environments, constructing a single task model based on task analysis is not sufficient. The device and context set their own constraints, which must be noticed in order to provide the user a usable sequence of actions. While it is rational to use task analysis to extract the main features of the task in order to form a general task model, this general model may require adjusting; some types of devices require a task model of their own. An inflexible task modelling mechanism in multi-channel environments results in bad usability, at least for some user groups of particular devices. For the rest of this work, this problem field is referred to as the "device-task problem".

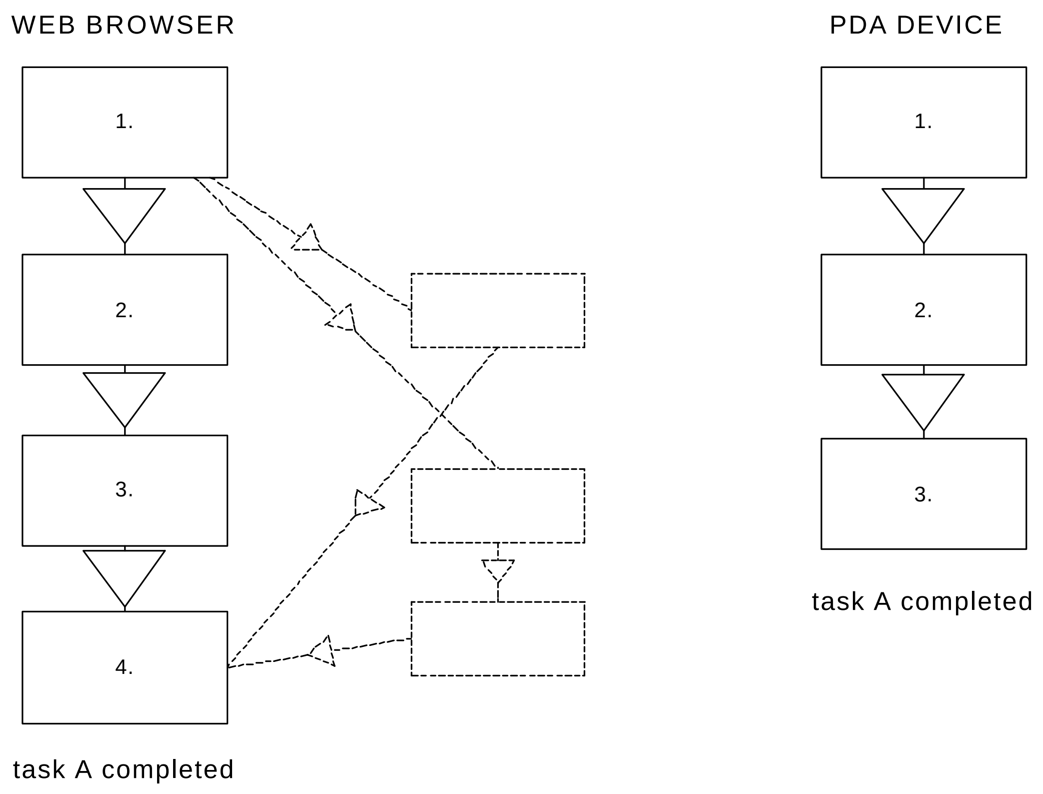


Figure 1: Completing a task through diverse sequences of actions.

It is possible to try to automatically fragment the information that is sent to a small screen device into several small pages while providing a navigational structure with a link to each of the pages [Eis00]. This usually does not eliminate the navigational problem however, but only increases it by easily disorienting the user [Sch01]. For a small screen device, the essential content must be accessible without additional “machine reasoned” hierarchy in the navigational structures. It clearly seems that the used UI development mechanism should free the designer to describe the task flow in a device specific manner.

3 Device Independent UI Languages and Models

Developing user interfaces is expensive and laborious. In systems with graphical UIs, nearly 50% of the source code and development time can be related to the UI [Mye92]. In systems where multiple types of user agents interact with the server, the problem is bigger and may become overwhelming. For enterprises, it is financially and maintenance-wise almost unbearable to provide multi-channelled services by using ad hoc methods.

3.1 Device Independent User Interface Languages

In near future, the growing variety of Internet devices makes enterprises more reliant on device independent user interface languages. With these languages, content can be delivered by using a single user interface description, which is automatically transformed into appropriate formats for the various types of devices, as is shown in figure 2. A transformation module must be associated with each language to provide interpretation for the abstract UI description, and to transform it to other devices accordingly. Because of the advantages discussed earlier, XML has already been used as the basic technology for some UI languages. In the following sections, a number of these languages are briefly presented.

XUL

XUL (XML-based User Interface Language) [XUL99] can be used to describe most of the elements that are found in contemporary user interfaces, such as buttons, toolbar-components and popup-menus. The language was originally developed to facilitate building the user interface for Mozilla browser. XUL does not address reactions to user interface events, such as user pressing a button. In addition, no abstraction considering user interface elements is available [Mül01b], thus XUL is essentially a device dependent language.

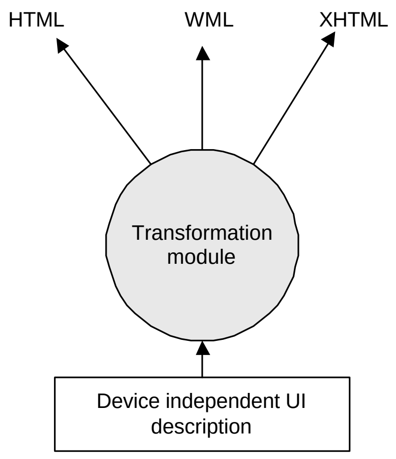


Figure 2: An abstract UI description can be transformed for various target platforms.

UIML

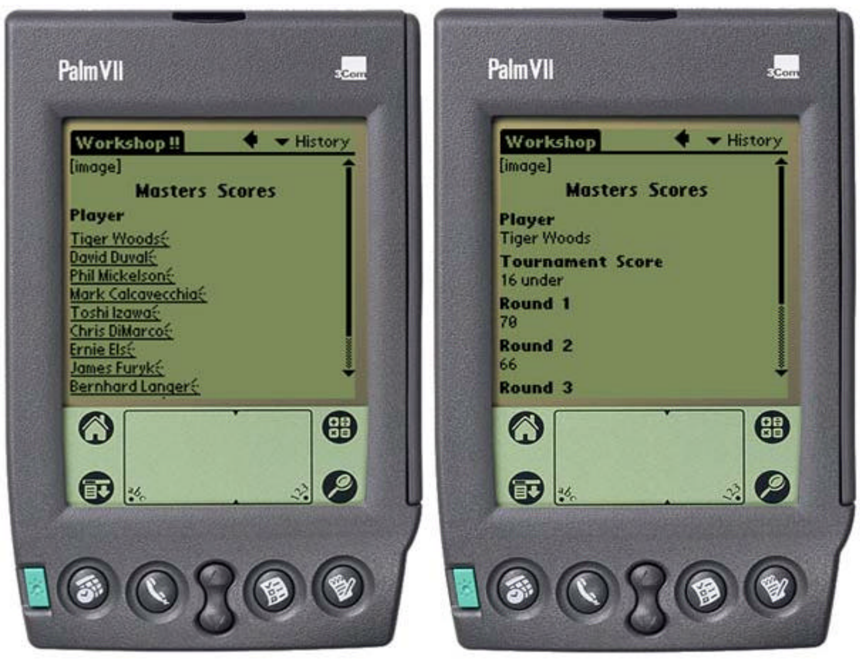
UIML (User Interface Markup Language) enables user interface descriptions in a level similar to XUL, but its elements have abstraction, which makes UIML interfaces essentially device independent. UIML also contains an event-handling mechanism to address communicational issues between user interface and the underlying software. A separate rendering module is required for each target device that uses the UIML-based application. This is considered a major problem [Mül01b], because the true functionality of the UIML-based user interfaces is always dependent on the implementation of this module, called renderer. An equally important flaw is that selecting target specific features is limited due to the lack of appropriate mapping concept [Mül01b] (see chapter 3.2). This suggests that achieving true device independence with UIML might be impossible.

MAXML

MAXML (Multi-Channel Access XML) is a language developed by Curious Networks [MAX01]. It distinguishes from previously introduced languages in that it is specifically designed for use with a user interface engine called Continuum. The Continuum engine is capable of converting the MAXML-based UI descriptions into various target languages. In addition, the engine is capable of handling the actions triggered by a user agent. A demonstration presented by Curious Networks, shows how a simple user interface description is transformed for a number of devices, including a Web browser, Web-enabled phone, Windows CE device, voice browser and a Palm VII device [Cur01]. The UI is rendered for all these devices from a single MAXML description, which is not presented here because of its length.

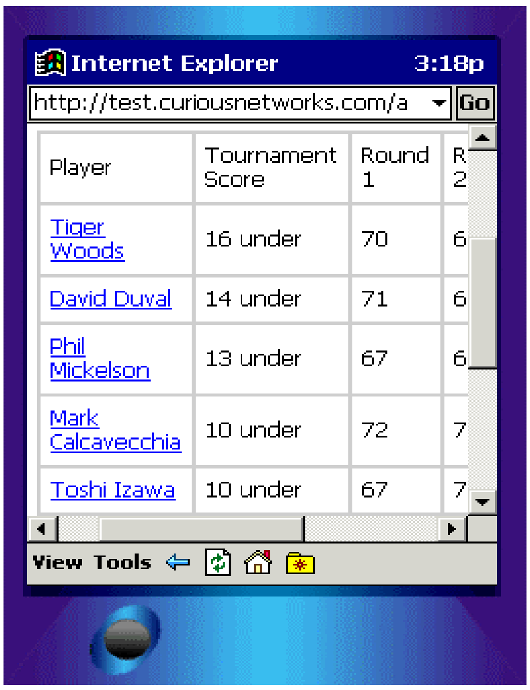
Figure 3 shows an example of how the Continuum engine views golf results for a Palm VII device. The view is automatically split in half: the column "player", which is specifically marked to be a searchable column (a mechanism provided by the grammar), is automatically transformed into link, as the framework notices that the browser is a Palm VII. Through player links, the user is given the possibility to view each players results.

Figure 4 presents the same view rendered for a Windows CE-device. Importantly, the target device seems to have an impact on the way that the engine produces the navigational structure of the UI. The information that is divided between two screens for the Palm device is now viewed in a single screen. For some reason the UI engine still presents the player names as links although the rest of the data is shown already. It is not explained what happens if the user clicks the player name, but it can be assumed that information similar to figure 3b is presented. The Continuum engine seems to contain hard-coded logic for making navigational decisions on behalf of the designer. To some of these decisions the designer cannot affect, even when that would be necessary. Consequently, the usability of the resulting user interfaces may not always be acceptable.



a) Players are links to results b) Result view

Figure 3: Golf results viewed by a Palm VII device.



Player	Tournament Score	Round 1	R 2
Tiger Woods	16 under	70	6
David Duval	14 under	71	6
Phil Mickelson	13 under	67	6
Mark Calcavecchia	10 under	72	7
Toshi Izawa	10 under	67	7

Figure 4: Scrolling violates usability.

Figure 4 shows a usability problem in the rendering of the use case for the Windows CE-device: the user is forced to horizontally scroll the interface to be able to see the data. This particular problem may not be a significant one, at least not for all users, but it does bring to mind considerations about further usability problems. The use case that is shown here is a simple one. It can be assumed that when applications get more complicated, more severe usability problems can occur.

pXML

pXML has been designed as part of OpenPort framework development at SysOpen Plc [Sys01]. Of the three languages discussed earlier, pXML bears closest resemblance to UIML due to similar abstraction level and the visual-oriented starting point. In this section, we discuss the motivation for launching the OpenPort project and along that, development of the pXML language. Discussion of the more specific features of pXML is tied to presentation of the RDIXML language in chapter 4.

Modern Web applications are strongly data-oriented and therefore a means to describe the flow of the data between the UI and the server is needed. In order to achieve a satisfactory system wide solution, the user interface definition mechanism must provide a comprehensive solution to tie the UI functionality to the rest of the architecture. Otherwise, the UI remains an element too separate from others, and does not offer a satisfactory communication interface towards the rest of the system. Failing to address this issue easily leads to bad overall solutions that especially might result in bad performance. It is an important part of the UI functionality to describe the interactions with the client and the user. For example, the required information includes describing operation flow, parameters, and data retrieval mechanisms.

Script languages such as JSP and ASP have been developed to cover these kinds of requirements. Use of these technologies comes at a price, however. Using them requires additional knowledge on how to divide work among developers. It is tempting to build UIs by intermingling UI design with technological details. Successful use of these technologies relies on awareness of best practices and common design solutions [Alu01, p. 30]. In addition, device independence is not built into these technologies, although technical remedies for this problem have been suggested [Sun01]. If device independence is the goal, a device independent user interface language is needed independently of the platform technology used. Both JSP and ASP are essentially template-based technologies, meaning that a separate user interface implementation is required of each specific device.

Both technologies require combining device specific markup with traditional programming.

The data-oriented problems mentioned above together with the need for an effective mechanism for developing device independent applications were the essential reasons for launching the OpenPort project. The pXML language directly integrates to the underlying OpenPort framework by providing means to describe communication with deployed application objects. For example, pXML lets the designer model the data management objects including their operations and attributes. The data elements work directly as a basis for the functionality of the data management module of the framework, i.e. names and types of database fields are not hard-coded into the EJB-bean source code. High-level data retrieval and processing can be managed by using elements of the pXML language.

Although pXML reuses elements in both data and UI descriptions, it embeds the interaction descriptions between the client and the server into presentation descriptions. Some tests have been made, where the same pXML-based UI description has without modifications worked for both Web and WAP browsers. However, successful usage of a similar presentation structure for these two types of devices is rare. Although the OpenPort framework supports adapting content to different kinds of devices, this may require separate pXML implementation for almost the entire UI. Binding interaction descriptions to presentation elements decreases the level of code reuse between device specific user interfaces.

3.2 User Interface Modelling

Various XML-based languages have been proposed as suitable for device independent user interface development. It seems, however, that these languages operate in an abstract level too high for the successful producing of UIs for different kinds of devices. None of the languages mentioned in chapter 3.1 notices the differences in navigational requirements between various devices. The pXML language allows the designer to adapt content to device specific requirements, but this decreases the level of UI code reuse.

Importantly, most of the languages are not task-based and hence do not support transforming content to voice-based browsers, thus failing to meet the requirements of pervasive computing (see chapter 5.2). The faulty starting point and the lack of appropriate mapping mechanisms and modelling structures makes it difficult, and sometimes impossible, to successfully implement device independent user interfaces. However, because of the high demand for multi-channel content delivery and the benefits related to higher abstraction, many enterprises are eager to use device independent UI languages. If the expressive power of these kind of abstract languages is insufficient for producing usable interfaces, many Internet applications will in future end up not being all-inclusively usable.

User interface modelling (UIM) systems provide an alternative solution to UI development; UIM languages consist of models for expressing the various properties of the user interface. UIM technologies aim to provide an environment, where UI development and implementation is easier and happens in a more professional and systematic way compared to traditional UI development tools. To achieve this goal, user interfaces are described by using declarative models. Typically, model-based systems are clearly task-oriented, i.e. modelling the tasks of the user is the basis for the development of the other models.

An example UIM system is TEALLACH, which is developed specifically for work with object databases [Gri99]. TEALLACH divides the various elements of UI into task, domain and presentation models. The domain model is used for modelling structural and behavioural features of application objects. The task model is used for modelling user tasks, as well as associating domain elements into tasks. An abstract presentation model is used to give the tasks and domain elements a general presentation, which can be specialized with environment specific UI elements. TEALLACH does not aim to contribute in the area of individual models of UIMs, rather it emphasizes facilitating combining the various UI elements in an effective and systematic way. A model-based user interface development environment (MB-IDE) is presented to facilitate combining the three essential UIM models to achieve concrete user interfaces.

Defining a modelling structure for a language results in three major advantages listed below [Sil00]:

1. User interfaces can be described in a more abstract manner
2. Models facilitate developing methods for more systematic design and implementation of user interfaces, because they provide potential for
    a. modelling user interfaces by using various abstraction levels,
    b. incremental development of the models and reuse of the user interface descriptions.
3. Models provide the foundation for the automation of the tasks related to design and implementation of user interfaces

UIMs are expressed in some kinds of formal languages, which address the various issues of the UI by defining a grammar for the conceptual model structure and elements of the UI. By putting emphasis on structure, UIM languages enable separation of abstract UI elements from concrete ones. Typically, UIMs also introduce information that is not present in traditional UI languages, such as capabilities of various user devices. The clear separation of conceptual models at syntax level makes it also possible to dynamically provide interpretation and mapping for the various elements of the UIM in a more profound level, i.e. the run-time adaptability of the user interfaces improves. In addition, model structured languages suggest new development strategies; for example, a specific visual part of a user interface is usually not written out in one consecutive set of lines. Rather, the development process is carried out by filling the elements of separate models, which are then dynamically mapped together at run-time.

The languages presented in chapter 3.1 do not have a clear model structure. Even though the pXML language, for example, does clearly separate models at some level, the separation is not consistent. The potentially abstract elements of the UI are mixed with concrete ones and thus it is not possible to separate the UI elements that are common for all devices from those that are not. This results in lack of ability to successfully reuse elements of a single UI description for devices that are significantly different from each other.

The UIM systems and their notations often prove to be complex especially when it comes to learning and using them [Sil00]. Various case tools have been developed to help overcoming these complexities [Pue99a], focusing mainly on describing model data and combining the models to each other. These tools are not in the scope of this thesis, however.

Components of User Interface Modelling

UIMs consist of several declarative models describing the various aspects of the UI. Each aspect partly affects the dynamic process of producing concrete user interfaces. Some essential models are presented in table 3. The used models vary according to each UIM and the ones presented in the table are collected from various sources. Each model is only given a superficial glance, as especially modelling tasks and devices form research areas of their own and are far too extensive to be well covered in this thesis.

Model name	Description
Task model	A task model describes how users do their tasks in a certain application. It contains the task structure, and the order and division of interactions between user and system [Pue99b]. It can be said that a task model describes the static and dynamic organisation of the work.
Dialogue model	Dialogue models contain such information as to which objects exist in the user interface and what are their possible states. The actions that the user may initiate through the user interface, as well as the reactions that the application may execute via those elements, are represented [Pue99a]. Some models combine tasks and dialogues into one model, which is usually called a task-dialogue model [Sil00]. Dialogue model can be seen as a more concrete approach to task model.
Domain model	A domain model defines the underlying objects that the user can indirectly see and manipulate through the user interface. In addition to application's data model, it is also intended to explicitly represent the attributes of the

	object and to express the connections between various objects [Pue99a].
User model	A user model defines the attributes and roles of users and it can be used to provide a way to model UI preferences for specific users or roles [Pue99a]. However, user models are described vaguely in literature and are present in very few UIMs.
Presentation model	Presentation model represents the visual, haptic, and auditory elements provided to the user by the user interface. The presentation model is basically just a static collection of sensory elements [Pue99a], but attaching stylistic properties, such as colours and font size, to the user interface is also considered to be a part of this model.
Device model	Device model presents the capabilities, such as the used UI language, connection speeds and other properties of the device [Mül01a]. CC/PP (Composite Capabilies/Preference Profiles) is an existing standard to do exactly this [W3C00a].

Table 3: Essential models of UIM systems.

Levels of Abstraction

User interface models are twofold in nature. On one hand, fully abstract elements such as user tasks are always present. On the other hand, some elements are concrete and hence can be only in some specific UI environment. An example of a task, i.e. an abstract element, is paying a bill. This task consists of a group of subtasks, which when executed in certain order, complete it. These kind of abstract elements of the UIM, which are often called abstract interaction objects [Van93], use implementation that is not dependent of any platform's application source code. In addition, they are not executable in any platform and do not restrict further implementation, and are thus completely portable [Pue98, Mül01a, Mül01b]. For instance, task and domain models are usually thought of as being this kind of abstract elements of any UIM [Pue99a]. Their implementation is dependent on the mappings that combine them to concrete elements of each UIM.

The visual and auditory elements that the user can see or hear and manipulate on the user interface (e.g. menus and pushbuttons) are concrete elements, which have to be defined unambiguously. At least presentation and dialogue models are usually classified as being models that are concrete in nature [Pue99a].

According to Puerta and Eisenstein [Pue99a], there commonly exists a mismatch of abstraction levels in the developed UIMs. Because the UIMs have both abstract and concrete elements, certain properties and attributes have to be defined in the abstract models in order to map them with the concrete models. Failing to address the necessary mappings results in an inflexible UI development environment. This problem, which relates to bridging the gap between the abstract and concrete elements of the UIM, is called the mapping problem [Pue99a].

Many UIMs restrict setting mappings and use automated processes to combine relatively simple structures. This seems to restrict freedom in design, and UIM techniques have been criticized for their inability to enable expressing personal taste and richness in user interface design [Pue99a]. Each UIM should serve the needs of enterprises and designers to show innovation and creativity in the user interfaces of their applications, as no user interface modelling system aims to force each user interface to function and look alike [Sti98].

First generation UIMs did not contain abstract user interface elements. Concrete elements, such as layout and widget customisations were involved right at the beginning of the development work of each UI [Sil00]. This was a major obstacle to generating device independent user interfaces. Second generation UIMs, such as TADEUS [Elw95] and MOBI-D [Pue97] enable user interface description in a higher level of abstraction. Still, a concrete proof of UIMs with mappings flexible enough and feasible as well as comprehensible has been missing, although research has been done in this area [Pue99, Mül01a].

Combining the Model Components

As Puerta and Eisenstein state: “if for a given user interface design it is potentially meaningful to map any abstract interface model element to any concrete one, then we would probably be facing a nearly insurmountable computational problem” [Pue99a]. It is thus important to try to discover the most intrinsic and beneficial mappings, through which it would be possible to proceed to desirable kinds of user interfaces in an effective and systematic manner. Figure 5 presents the problem domain: it is hard to know, which models should be combined and how.

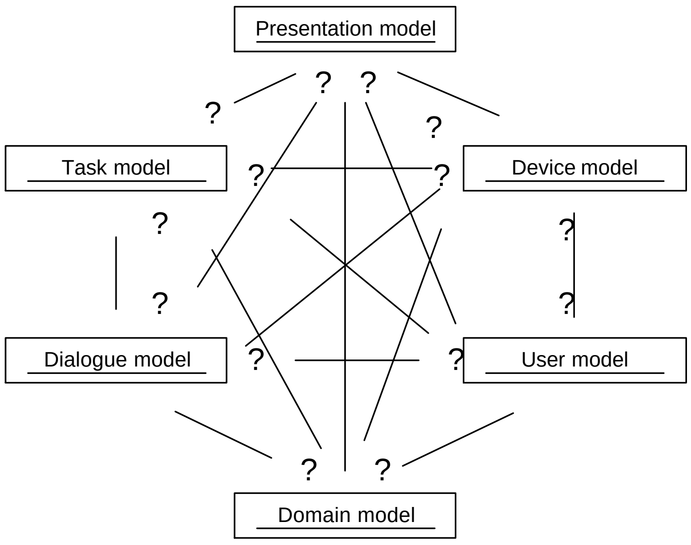


Figure 5: The mapping problem.

Examples of mappings that are considered important between models are briefly presented in table 4. The mappings are mainly described as presented by Puerta and Eisenstein [Pue99a] and do not cover all possible cases. For example, the mappings between task and device models are not presented, but are covered later in chapter 4.

Mapping	**Description**
Task-Dialogue	Task models describe the static and dynamic organization of the user's work. For instance, conditions on the flow of the work might be presented. Dialogue models define conversational activity between the user and the interface in a more concrete manner and thus enable or disable certain user interface functionality. This leads to parallelism between the models and naturally motivates mappings between them.
Task-Presentation	Users accomplish tasks through a user interface. As tasks in task models are expressed in an abstract format, it is natural that defined tasks should be mapped with some sort of visual or auditory presentation that enables users to execute given tasks.
Task-User	One task may need to be executed differently based on user requirements. It may therefore be important to map user data to the task being carried out in order to provide user-centered task flow.
Task-Domain	Performing a task involves manipulating domain specific objects. Mapping the workflow of the task model with contents of domain model makes user's tasks meaningful.
Domain-Presentation	Domain model elements contain attributes that affect the object presentation. Naturally, the presentation of pictures must differentiate from presentation of text. Therefore, it is natural to map a domain object to a presentation model based on its characteristics.
Presentation-Dialogue	In order to specify a running user interface, the elements described in presentation and dialogue models must be linked. The presentation model describes the actual components, which give user the tools to communicate with the software as presented by the dialogue model.

Table 4: Some of the possible mappings between UIM models.

Mappings can be done in two ways. The UIM language can provide the designer tools for the mappings, but it is also possible for the run-time engine of the UIM system to do the mappings automatically. Preferably the language should provide possibilities for a wide range of mappings, and the run-time engine should then automatically do only those obligatory mappings that were left undefined at design time. This kind of strategy gives design power for the designer, as well as eases development in cases where advanced design is not necessary.

4 The RDIXML Language

This chapter presents the objectives for the RDIXML language, as well as discusses the solutions, i.e. the models and mappings between them, in a conceptual level. The actual DTD-based grammar of the task model is presented in appendix A.

4.1 Objectives for the Language

There are some very general objectives that should be recognized in the development of any UI language. Following the objectives set by Eisenstein et al. [Eis01], the RDIXML language should be

?? **Declarative**
?? **Comprehensible to humans**
?? **Formal so that it can be understood and analysed by computer systems**
?? **Platform independent**

To get over some major difficulties in multi-channelled application development, additional objectives are set. These are expressed in the next few paragraphs.

Device independence – RDIXML is built for use in multi-channel environments. Therefore, the language must provide means to build UIs without direct integration to any specific UI environment. This objective should not conceal the fact that some devices require specific implementation details. The language must be capable of expressing device specific features, but these expressions must not be expressed with device specific markup.

Adaptation to device specific requirements - The objective of making applications universally usable imposes recognizing the differences between various UI environments. The presentational aspects of the UI must be adapted to meet the device specific requirements. The same applies for the dialogue and domain elements. In addition, it is now clear that the steps that must be taken in order to complete a task are not always equal between diverse devices. Providing possibilities to alter the flow of actions is the key for all-inclusive usability in multi-channel environments.

Task element reuse – Device independent user interface languages are piquant since they enable content distribution for many different channels based on a single UI description. Yet, the functionality of some devices is so different that clearly some elements of the UI require device dependent implementation. This is the case even with user's tasks, which in the UIM literature are ubiquitously regarded as being directly portable between devices and hence reusable. From solving the device-task problem, it follows that the language must be carefully structured to reuse common task elements.

Direct integration to the underlying framework - A user interface bridges humans and machines by allowing the exchange of information [San01]. Not fully following the spirit of this definition, UIM systems have rarely been associated with database environments [Gri99]. In general, it is considered easier to create new services based on existing components of the framework, if the language directly bridges UIs to existing technologies, such as various data services. RDIXML aims to directly integrate to the underlying architecture, i.e. its elements should work in parallel with the functionality of the RDIXML framework (see chapter 5). This allows direct communication with various databases, as well as use of other services provided by the framework. This is important in the sense that as software management gets more and more complicated, there's an urgent need to utilize pre-existing services, through which application functionality is mainly achieved by specializing the framework and not by modifying the core of existing program code [Voa98].

The key for fulfilling the set objectives is clearly first to *find the necessary models* and *model them in an appropriate way*, and then *solve the mapping problem* between the chosen models. For the rest of the work, the single word RDIXML always refers to the developed language and the word framework is used to refer to the RDIXML framework.

4.2 Models of the Language

RDIXML has a modelling structure consisting of task, domain, user, dialogue, presentation, device and application models. These models do not cover all issues related to UIs in the Internet world, such as user's location and personalization. However, they suffice to sketch the basis for the language; additional models and functionality can be added in the future. Each model is discussed in terms of its functionality and main characteristics, including a brief example. Further examples are given in chapter 6. The model structure of the language and the ways that the models relate to each other unavoidably predetermine the interpreting requirements for the RDIXML framework. Hence, some architectural points of view are considered already in this chapter. The task model, being the cornerstone of the language, is given wider attention compared to other models.

Except for the task and device models, the pXML language has clearly affected the structure and elements of RDIXML. However, RDIXML is essentially task based and thus the "emulated" elements in many ways have a different character. The focus in presenting the models of the language is on the most interesting task-related elements. In the scope of this work, none of the models of the RDIXML language is fully completed.

Task model

The task model is a key construct in the RDIXML model structure and clearly the starting point for the design of other model elements. Indeed, the success in the task model construction no less than directly affects the success in fulfilling the objectives of this thesis. However, task modelling is a difficult issue in model-based user interface design, and not all aspects of it can be addressed in the scope of this thesis. For example, defining relations between tasks, especially for multi-channelled applications, is a complex issue. The RDIXML task model is a simplified one leaving many generally important modelling aspects without attention. The focus is on finding ways to reuse the task model code, while allowing device dependent task descriptions. Table 5 presents the central elements of the model.

Element/attribute	Description
Task	Identifies the user's high-level goal.
subtask	Subtasks are used to describe the user's lower level goals, i.e. the phases of actions that must be completed in order to reach the higher-level goal.
Precondition	Preconditions define the tasks that must be finished before the task in question is enabled.
Postcondition	Postconditions define conditions and actions that are triggered at the realization of the condition. Through postconditions a task may invoke or disable a number of other tasks.
Systemaction	Systemactions describe the data management actions that the system must take in order to be able to present the user data related to the task. Systemactions exploit operations modelled in the domain model.
Useraction	Useractions are used to describe the management of actions that users trigger from the UI. Useractions exploit operations modelled in the domain model.
Fieldreference	Fieldreferences refer to domain model attributes that are associated with the task. The fieldreference elements abstractly define whether the attribute should be given a value by the user (input) or should just be viewed to the user (read-only). A similar concept is used by Müller et al. in [Mül01a].
Navigation	The navigation element forces the designer to define navigational properties to the dialogue model. This functionality is connected to the framework's validation system (see chapter 5.4), i.e. the validation system will raise an error if the navigation is not provided.
Tasknavigation	The tasknavigation elements allow postconditions to force dialogue level navigation to some other task. Hence, the element forces the presentation designer to maintain task flow at presentation level.
Systemtask	This element is used to define reusable systemactions, which can be referred to from within separate tasks.

Table 5: Key elements of RDIXML task model

Chapter 2 showed that device specifically designed task flow is sometimes a prerequisite for a usable UI solution. Now, considering the task description reuse, this has serious implications: the preconditions and postconditions that are related to a task inevitably become device dependent. For example, if a user completes task A with a regular Web browser, this might lead to invocation of two alternative ways to proceed with the task, i.e. the postcondition of the task defines that two tasks must be invoked. Completing the same task with a more restricted device might lead to invocation of one single subtask that the user is allowed to execute. Hence, the postconditions for a single task can be device dependent. Following similar logic, the same applies for preconditions.

Quantitatively, the largest part of task modelling consists of those elements that are mapped to domain model. These consist of fieldreference, systemaction, and useraction elements. It would seem that these elements are more commonly portable compared to condition elements. Even if the task flow might change, it is often the case that same business operations and attributes can be used. This is not always the case, however. The workers at SysOpen plc have recognized that business methods are sometimes more manageable if they are called device dependently, i.e. a separate method is reserved for each device type. The device can affect the amount of data sent to the user interface, as well as other properties of the functionality. Hence, the domain related task elements are in some cases device dependent, especially concerning operations. This, together with the discussion considering task conditions, suggests that rather than tasks as whole, some parts of them are device dependent.

While most UIM systems do not seem to address device specific task management, Eisenstein et al. suggest that mapping tasks to devices can solve the device-task problem [Eis00]. Their mechanism separates task flow into device dependent entities, while it does not address the consequences considering task reuse. We claim that simply mapping tasks to devices severely decreases the level of UI code reuse as it imposes completely separate task implementations for differently behaving devices. Separate task implementation suggests copying the common task elements between device specific task versions. Obviously, reusing smaller task entities enhances the UI implementation process, as it does not force the designer to work in a “copy-paste” design environment. A more fine-grained mapping mechanism is required between tasks and devices.

Previous paragraphs show that separate task model elements may have relations to various devices independently of each other. RDIXML addresses this problem by categorizing separate task elements according to device groups. Essentially, this means that although the common elements of the tasks remain portable, several elements inside a task description can be device dependent. For example, considering an otherwise portable task, some specific device could set a different precondition for activation of the task. The rest of the work refers to this concept as the *device categorized task model* (*DCTM*) of RDIXML. For a more detailed view, see appendix A.

Presentation model

The presentation model defines the layout for the UI and comprises of three elementary elements: views, subviews, and layout structures. Subviews define autonomous entities of a user interface. For example, a single subview might contain a website's navigational structure, which might be reused in combinations of several different subviews. Layout structures define the view layout, i.e. the size and division of layout elements, as well as their relative positions. Views define combinations of subviews by mapping them to specific slots in a layout structure. Figure 6 visually presents these elementary components and their relations.

Each subview potentially contains a number of tasks. Subviews define presentation for the tasks, i.e. for the attributes and actions related to tasks, as well as their positioning at the screen. A sketch of a possible RDIXML syntax for such presentation is presented in figure 7. In the example, a domain attribute "name" is given a label. The actual name is to be printed in an "editbox", which is an abstract presentation type definition. The type suggests that the user may edit the value of the name. Hence, the corresponding task element must have defined the attribute name being of type "input" (see "task model"). A device dependent rendering module must transform the abstract presentation type "editbox" into a suitable markup. For a regular HTML browser, the output of the attribute could be '<input type="text" name="name" value="x"/>'.

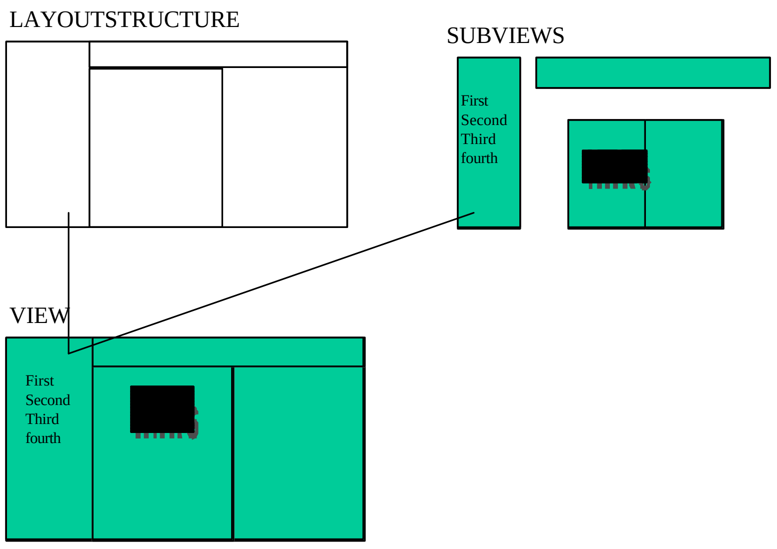


Figure 6: Views combine subviews by mapping them to layout structures.

A general notion related to abstract UI languages is that describing UIs with device dependent languages, such as HTML, allows direct expressions of platform specific presentation details. This is not possible when using abstract languages such as RDIXML. The transformation from abstract UI elements into device specific markup is based on programmatically predefined interpretation model, to which the UI designer can affect only so much. Thus information concerning small details, such as exact locations of objects, or producing stylistic layouts by using complicated table structures, may sometimes prove to be impossible to express with abstract level languages. Usage of device specific style sheets, such as cascading style sheets (CSS), can sometimes help to overcome appearing problems. However, this may result in troubles with the style sheet handling capabilities of various browsers. On one hand, extending the RDIXML grammar to support additional features could help answering to device specific needs. On the other hand, this would probably severely hamper the overall intelligibility of the language. The problem is important and interesting, but too extensive for the scope of this thesis. Hence, we content ourselves with proceeding with general requirements.

```
<fieldproperty reference="name" label="Name">
    <UIcomponent type="editbox"/>
</fieldproperty>
```

Figure 7: Presentation structure for a single attribute.

The expressive power of the presentation model must suffice to define common presentation formats and components. For example, radiobuttons and checkboxes are used practically in every UI environment. Another example of a general need is that alignment requirements are commonly satisfied by embedding the content into some kind of a table structure. Hence, RDIXML should be capable of unambiguously expressing these kinds of presentations. The language must be accurate enough to enable transformation modules to make a satisfactory transformation into device specific markup. Yet, the expressions must be abstract enough not to blur the transformation logic for devices with limited capabilities.

An example of presentation implementation is given in appendix C. The example relates to the simulation presented in chapter 6.

Dialogue model

A significant starting point for the design of the language has been to define a clear role differentiation between task and dialogue models. The two models describe the interactions between the user and the server, but in different abstraction levels. As Bomsdorf and Szwillus state, the division of elements between these models is indeed a difficult separation to make [Bom98]. Task models should describe the interactions in those parts that clearly relate to users goal, whereas the dialogue model refines the interaction to meet device specific requirements in technical sense.

A business rule saying that users must be authenticated before accepting their orders is clearly a rule that must be obeyed independently of the device used. This kind of a rule clearly has to do with general application requirements and should be implemented with elements of task model. On the other hand, if a device supports scripting, form input can be validated at the browser environment, whereas in the case of non-scripting UI environment, the validation is left for the server. In the latter case, the server does the validation and it must return the form in case of faulty input. The script usage presents a technical difference in the interaction model between two devices, this having nothing to do with the user's task. The problem with the validation, namely whether to use scripting or not, is exactly what dialogue modelling is aimed at: describing interactions for the sake of technical differences. Obligatory technical changes in the interaction model motivate the existence of the dialogue model.

To enable technical interaction enhancements, the dialogue elements must be able to override subtasks descriptions of task elements. Sometimes, the overriding could happen by using scripting functionality. Importantly, scripts must commonly be activated when the user interacts with the UI and thus they must be attached to certain presentation elements of the UI. For example, some HTML-based devices can effectively combine usage of framesets and javascript, i.e. they define the view functionality by combining presentation and dialogue elements. The presentation and dialogue models are very closely tied together and, for this reason, we suggest merging these two models. In most cases, it can be assumed that if the presentation changes, so does the scripting functionality. This solution may on some occasions decrease UI code reuse, but in most cases is a more comprehensible solution. The presentation model in appendix C contains some notes on how dialogue model elements could be used related to that specific presentation.

Domain model

The domain model publishes fields and operations of domain objects, i.e. it reflects the properties and capabilities of objects in the data management tier. By defining object attributes and their data types as part of the language, a checking mechanism for the integrity between the presentation and task elements, which relate to the domain elements, can be built. In addition, by using advanced programming techniques, namely reflective features of programming languages, data objects can directly use the XML-based domain definitions as their attribute definitions (see chapter 5.3). Hence, there is no need to hard-code attribute names with programming languages; a more maintenance friendly solution is to describe them as part of the domain model. OpenPort framework has successfully used this technique (see chapter 3.1).

An example of the possible syntax for the definition of a domain object is presented in figure 8. The element *domainobject* defines a domain object with attributes, operations and keywords. The basis for automatic rendering of domain elements is given by publishing the data types of domain objects. For example, if the data type of an element is picture, the automatic adaptation functionality of the framework will by no means try to embed the picture into a text field.

The example expresses the importance of an attribute with a reserved word "obligatory". For a device with limited screen space, unnecessary attributes can thus automatically be left out. Similarly, would the run-time environment of the RDIXML framework be capable of detecting changes in user's connection speed, unimportant attributes could be dropped out in case of low bandwidth. The *keyword* elements are used to match operation results with task postconditions (see "task management" in chapter 5.5).

```
<domainobject domainobject_id="orderBean">
    <fields>
        <field field_id="orderBean_productname" obligatory="false"/>
        <field field_id="orderBean_productid" obligatory="false"/>
    </fields>
    <operations>
        <operation operation_id="getOrder"/>
        <operation operation_id="submit_to_cart"/>
    </operations>
    <keywords>
        <keyword keyword_id="insufficient_credit_limit"/>
        <keyword keyword_id="success"/>
    </operations>
</domainobject>
```
Figure 8: A sketch of a domain object description in RDIXML format.

Device model

The purpose of a device model is to describe properties of various devices to enable adaptation of required content to meet device specific requirements. W3C has developed a standard, CC/PP, for modelling device properties and user preferences [W3C00a]. A specialized CC/PP profile can be used to guide the adaptation of content presented to that device. CC/PP is a meta-language that can be used to build vocabularies that can present the properties and capabilities of a certain device. Not many public applications of CC/PP exist, but an example is the WAP user agent profile (UAPROF) [WAG01].

As CC/PP is a part of a wider application, it doesn't cover all the issues that may be necessary for a given architecture [W3C00c]. For example, the CC/PP framework does not define the actions that UI systems should make to adapt the content to given device properties. Hence, the standard does not directly serve the needs of the RDIXML framework (see "automatic adaptation" in chapter 5.5). This work does not suggest how CC/PP should, or could, be combined with RDIXML to form a more comprehensive and useful device model. Instead, common device model properties that serve the needs of the RDIXML framework are presented.

RDIXML attaches domain attributes to tasks in an abstract format to achieve code reuse and device independence. Designers may sometimes wish to use the automatic mapping properties of the RDIXML framework instead of specifically defining presentation for these attributes (see chapter 5.5). Hence, it is important that the device model maps the abstract formats to concrete UI counterparts. For example, for each specific markup language, the device model gives a default correspondence for an abstract presentation type "input". At run-time, the UI engine can then map the given abstract data type to the default input widget. For example, for a regular HTML browser, the abstract data type "input" could be replaced with markup '<input type="text" …"/>'.

Importantly, browser manufacturers provide idiosyncratic implementations of language standards. It has traditionally been the case that browsers of Netscape and Microsoft have rendered the same HTML document in two different ways. For example, alignment of graphics has commonly resulted in browser specific rendering, i.e. the result often looks different on different browsers [Nie00, pp. 39-41]. It is not only the devices that need be modelled, but also the browsers and especially the differences between them.

It seems rational to think that for example two PDA devices with similar kinds of screen resolutions, user interface environments and connection speeds could use the same task descriptions, whereas a WAP browser perhaps should use another. Similarly, two PDA devices with similar kinds of screen resolutions could use the same presentation model, whereas a Web browser would most probably use another. For these reasons, it is important to model devices as a hierarchical tree, where devices inherit properties from upper levels of the hierarchy and are able to override them as necessary. This strategy supports property reuse, while it at the same time allows addressing minor deviations of idiosyncratic implementations. This strategy allows other RDIXML models to map to larger groups of devices at once, instead of separately mentioning each device. Figure 9 presents the overall structure of the RDIXML device model.

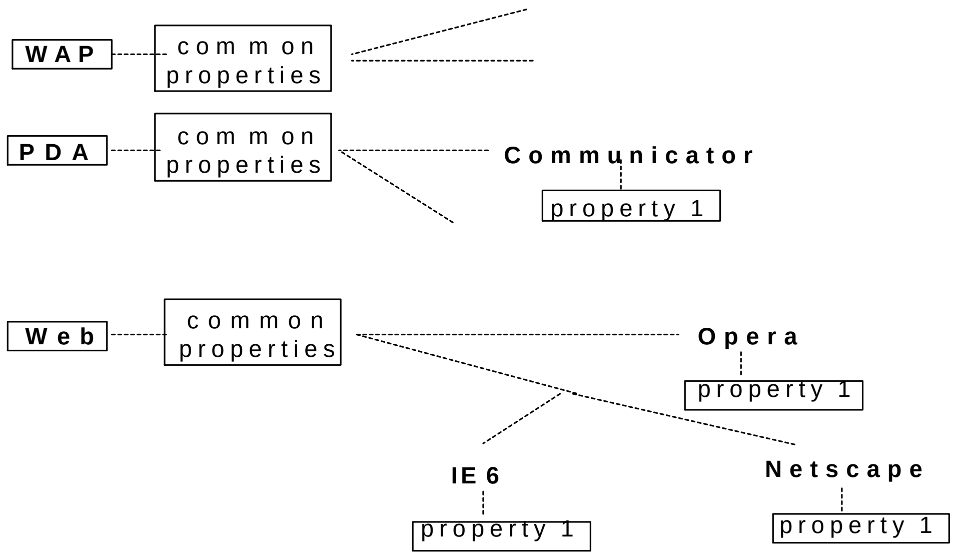


Figure 9: RDIXML models devices in a hierarchical tree structure.

The hierarchical device model would seem to provide a good basis for coping with the variance between markup language implementations. A good starting point would be to provide possibilities to simply model the usage of different tags or attributes to achieve the same functionality. For example, for some specific browser, tag <font-size> could be used instead of tag <font>, to express the font type. In more complicated cases, it would be of help if more complicated rendering rules could be applied device specifically. Specific device properties, such as screen resolution, are briefly discussed in chapter 5.5 (see “automatic adaptation”).

User model

A common requirement for an enterprise application is the provision of user or role specific content. Clearly, some tasks are shared by different user groups, whereas some are restricted to the more privileged users. On a more fine-grained level: the user’s role can affect the level of information viewed to the user, the visualization of the user interface, and the tasks that the user is allowed to perform. Some domain elements may not be viewed to the user and the ones that are might have a different kind of presentation and relate to another task.

As user privileges change often, publishing user and role information as part of text-based application logic, i.e. RDIXML descriptions, would not be a satisfactory solution. This would mean that the system administrator would then have to maintain user rights by changing the text-based RDIXML descriptions. Mechanisms that are more dynamic, such as LDAP (Lightweight Directory Access Protocol) [Wah97], are a better option for maintaining information about users and roles. Various enterprises use role names of their own and these roles have various hierarchical relationships to each other. Hence, the RDIXML grammar must define a mechanism to restrict and grant access rights to services in an abstract manner, without directly referring to any specific user or role names.

Because the RDIXML user model limits resource use with abstract constraints, the RDIXML framework must retrieve concrete user information from any service that might provide it. We do not specify how this should happen. Since the role maintenance information is not implemented with RDIXML, the principal task of the user model is to restrict the use of various other elements of the language. The elements of user model are thus embedded among elements of other models of the language. Figure 10 presents a simplified sketch of how task elements could be restricted. This matter is further discussed in relation to mappings between models in chapter 4.3.

```
<task task_id="buy_item">
    <userrule accesslevel="2"/>
</task>
```

Figure 10: Access level 2 is required for accessing task “buy_item”.

The RDIXML user model is essentially similar to the one of pXML language (see chapter 3.1). However, since RDIXML is a task-based language, the user model is also embedded as part of the task model.

Application Model

The application model ties model elements together to form a complete application; all elements of any file must unavoidably belong to some application. If many applications are parsed to RDIXML engine’s memory at the same time, the separately described model elements of the same application are bound to a single application namespace containing all resources of that application. The model must also define meta-data concerning various aspects of the application, such as starting points for the application for each device type. In addition, the model must list the files that applications comprise of. An example of application model contents is given in figure 11. Each RDIXML file must by definition start with element *application*. Apart from the other models, device model implementations are application independent and are maintained separately from other models of the language.

```
<application application_id=”test_app”>
    <defaultviews>
        <defaultview devicegroup="PDA" view="PDA_buy_item"/>
        <defaultview devicegroup="WWW" view="WWW_buy_item"/>
    </defaultviews>
    <tasks application_id="test_app"/> <!– placed in another file -->
    <views application_id="test_app"/> <!-- placed in another file -->
    etc…
</application>
```

Figure 11: Applications comprise of elements of several language models.

4.3 Model Mappings

The previous chapter already discussed mappings between models at some level. This chapter completes the picture by presenting all mapping relations that a designer may use to bind various model components together. There are no theoretical foundations to define the exact mappings between model elements [Pue98] and therefore applying them is more or less dependent of the capabilities of the designer and the objectives for the language. What is important for the UIM designer to keep in mind, though, is that several systems that fully embed the mappings into the programmatic code of the framework provide an inflexible UI design process. In these kinds of systems, the users of the framework cannot reach and affect the inner functionality of the framework [Pue98].

It is important to carefully divide the responsibilities of the model mappings between RDIXML and its programmatic interpreter, the RDIXML framework. In other words, the language should grant the designer tools to set necessary mappings manually, and the framework should be programmed to automatically make mappings left undefined by the designer. Hence, the RDIXLM mappings provide a tool for the UI designer to create accurate and personal design. Yet, a designer seeking for effective default solutions may leave some mappings for the responsibility of the framework. The automatic mapping process provided by the framework is further discussed in chapter 5.5. The RDIXML models and mappings between them are presented in figure 12. The application model is not presented in the picture, since elements of all models, except for the device model, are always mapped to the application model.

The figure clearly shows that the task model is the centerpiece, to which all other models relate. The upper part of the figure consists of several abstract level presentations (possibly containing dialogue elements), which are used according to the device detected. It is important to grasp the idea that the PDA view, for example, is an abstract description. This means that the view addresses aspects of the physical properties of the device, but is not bound to any specific markup. Hence, this view could be transformed to any markup (and thus reused), if the device in question would otherwise fit the properties of the view.

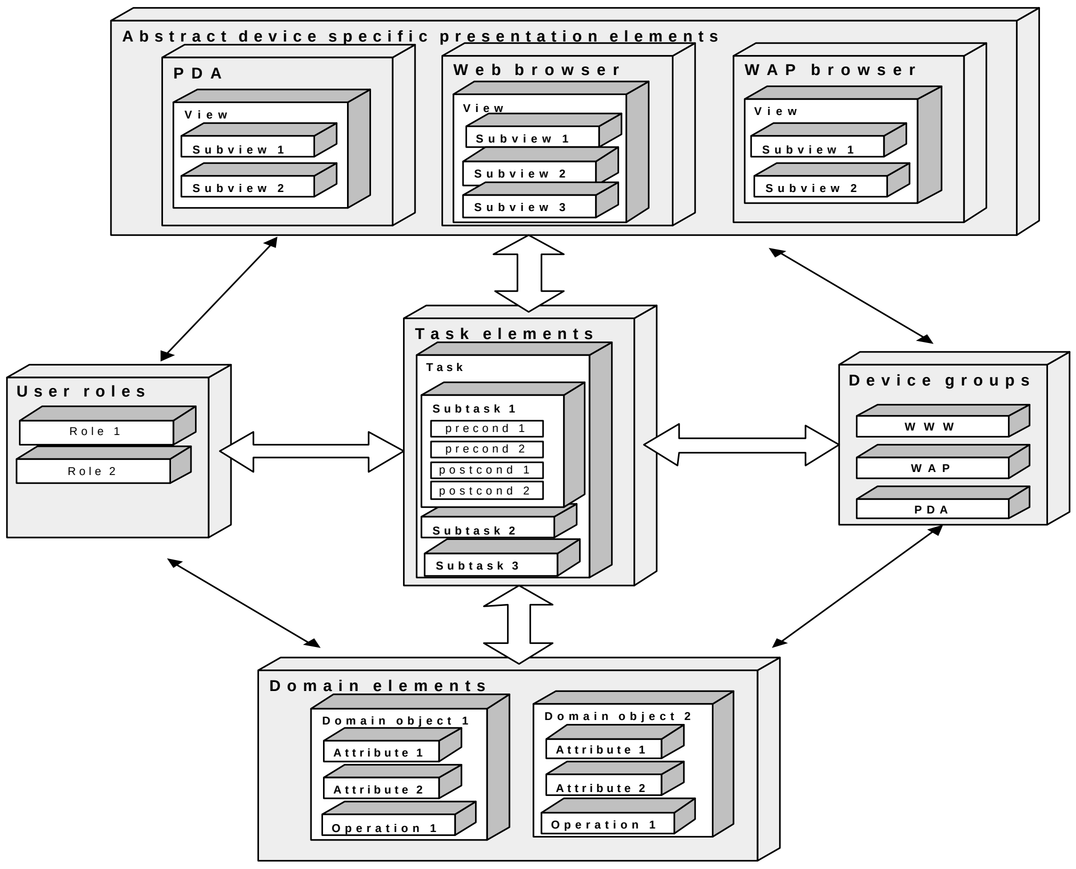


Figure 12: Mappings between RDIXML models

Table 6 lists each RDIXML design-level mapping. As every model of the language (except for the device model) must provide a mapping to some application, the application model mappings are excluded.

Mapping	Description
Task-Device	Task flow must sometimes be device dependent. At the same time, task element reuse is considered important. For these reasons, subtasks (as a whole), preconditions, postconditions, operations, and fieldreferences can be mapped to devices.
Task-User	Tasks can be user (or role) dependent as well as device dependent. The preconditions and postconditions of tasks must be mapped to users or/and roles. The user elements are directly embedded as part of the condition elements.
Task-Domain	Tasks provide a reusable way to describe bi-directional interactions between the client and server. To achieve comprehensively reusable UI descriptions, it is natural to map domain elements, namely operations and attributes, to tasks.
Presentation-task	The designers must have the power to define presentation for tasks. Hence, tasks are mapped inside presentation elements.
Presentation-Device	Tasks (and its subtasks) can be presented in many ways, because they can be executed by using various devices. This imposes a need to link presentations not only to tasks, but also to devices.
Presentation-Dialogue	Scripts must be attached to presentation components in a detailed level. In addition, it is often the case that when the presentation changes, so does the scripting. For these reasons, RDIXML presents a combined model for presentation and dialogue elements to provide a clear visual mapping to benefit the designer.
Presentation-User	Some presentation elements, namely views and subviews, can be user (or role) dependent. User elements are directly embedded as part of presentation elements.
Dialogue-Domain	Dialogue elements can override some definitions, namely operations, expressed in task models. Hence, there exists a natural mapping between the dialogue and domain models.
Domain-User	Some domain elements can be user (or role) dependent. Hence, the attributes and operations of a domain object can be user specific. The user elements are directly embedded as part of domain elements.
Application-Device	Complex applications may include functionality that is impossible to adapt to devices with limited capabilities. In these cases, applications must exclude certain devices or device groups. For example, an application that necessarily requires multi-media streaming functionality could not be used with a WAP device. It is thus necessary to specify applications in terms of their requirements [Ban00].

Table 6: Design time mapping possibilities between RDIXML models.

Chapter 3.2 presented three major advantages that result from dividing a user interface language into separate models. We state that RDIXML realizes language abstraction by being an XML-based language not directly relating to any specific UI environment. The second advantage (a more systematic UI developing method) is based on the various abstraction levels of the language models. The UI development is systematically based on reusing the abstract elements of the UI, namely the task and domain elements. The various mapping mechanisms and especially the device model properties realize advantage number three, i.e. foundation for automatic UI generation (see “automatic adaptation” in chapter 5.5).

5 The RDIXML Framework

The RDIXML language provides a basis for creating multi-channelled Web services. However, the language does not have an interpretation before it is provided by a software module that is programmed to tackle the grammatical presentations of RDIXML. Architectural components capable of interpreting the RDIXML language form a generic part of the RDIXML framework, which does not require changes on implementing new application services.

Proper comparison between various possible implementation strategies for this kind of a generic framework would be a huge effort. As the emphasis of this work is simply to prove the concept feasible, this chapter sketches one possible design and other possibilities are mainly left without attention. Chapter 5.1 generally introduces multi-channel architectures. Chapters 5.2 - 5.5 present different aspects of the generic framework particularly from the point of run-time functionality. Finally, chapter 5.6 discusses how the framework is specialized for application specific functionality.

5.1 An Introduction to Multi-channel Architectures

Software architectures describe the division of functional parts of application systems, as well as the predetermined co-operation of these parts. Good architectural design should produce systems that are efficient, interoperable, changeable, reusable, testable, and reliable [Bus96, p. 404]. A three-tier architecture model that is in wide use today comprises of presentation, application and data management tiers. The presentation tier is responsible for communicating with various devices; the tier handles requests from user agents and generates an appropriate presentation for the data presented to the agent. The application tier defines the application logic. The data management tier is responsible of managing application data in a stable manner. The two-tier model, still in wide use, merges the presentation and application tiers. Compared to the two-tier model, the three-tier model provides clear advantages; the most significant of these is the separation of data from its presentation.

Content is the valuable power source that makes communities, enterprises and individuals communicate via Internet. Dealing with the proliferation of various kinds of Internet devices requires new kinds of Web-architectures that can dynamically map the content to user interfaces that run on many different computing platforms. Most enterprises use independent systems for providing content for each separate channel [Lun01]. Economically and maintenance-wise, this is not a long-span way to act. Business processes should be streamlined to automatically provide content to and interact with many different channels, such as Web, voice, e-documents and digital television. "Enterprises that leverage the power of multi-channel content delivery will gain a competitive advantage over those that do not" [Lun01].

In multi-channel architectures, the role of the presentation tier is superior compared to single channel architectures. The tier must be able to detect various kinds of browser types, communicate with them with appropriate network protocols, and adapt the content to match the constraints of the device in question. As the presentation tier directly communicates with various kinds of devices, it must dynamically re-organize the UI components to adapt the content for various devices. The adapted content must be merged into a presentation format that suits for the target device; for example, the same content could be delivered in either HTML or WML formats. Equally, two HTML-based devices might require the same content with different kinds of presentation. This process may involve resizing UI elements and perhaps dropping out some non-obligatory elements of presentation. In addition, interaction sequences between the client and server may vary according to the device used.

5.2 Objectives for the Framework

The next paragraphs express the essential objectives for the RDIXML framework.

Multi-channel presentation tier - The framework must provide a run-time engine for managing the bi-directional communication with various user agents. The engine must be able to dynamically produce UIs for many different devices based on given RDIXML descriptions. Content must be adapted to meet the device constraints.

Task orientation - RDIXML is a task-oriented language and the framework must follow this premise. Founding application development on tasks, rather that assuming visual aspects of applications, is important considering goals of pervasive applications, such as enabling users to change the device on the fly [Ban00]. For example, returning to the office from the field, an employee might want to stop using the PDA device for an airline booking and rather finish it with the desktop computer. The graphics would change and maybe some alternative ways to finish the job would emerge, but the execution of the task would continue from the same point, provided that the devices could share session status. A task-based framework model very naturally seems to support this need: considering pervasive computing, it is important for the UI system to know at which state of task execution, not user interface, the user is [Ban00]. The remaining steps required for completing a task are what count, not the visual components (that are obviously not the same between devices). Importantly, a task-oriented system provides support for both visual and auditory devices. This kind of an approach enables saving the user's comprehensive task status on disconnecting. On returning to the service, the user could continue the service from exactly the same point where it was left. Task-based multi-channelled frameworks seem to open up new possibilities compared to traditional visual-oriented solutions.

Automatic adaptation – The framework should act as a complementary element to the RDIXML language and thus facilitate the UI development process. The framework should automatically provide mappings to help adapting content to meet device specific requirements. For example, if the designer does not define a presentation to a task attribute, the framework should automatically transform the attribute's abstract data type to some device specific presentation format.

Application independence - The framework's core must be a comprehensive solution neither requiring changes, nor new implementation for individual software projects. Achieving this objective requires a generic presentation module that is de facto reusable through every application project. Application specific UIs must be constructed by using the RDIXML language only, not by programming. Components of the presentation tier are generic in nature and cannot contain any application logic whatsoever.

In addition to these elementary objectives, common goals for application frameworks exist. Examples of such objectives are performance issues and support for application design. These issues are too extensive to be covered in this thesis, but are briefly discussed in chapters 5 and 7.

5.3 Technological Foundations

Building a fully functional application framework from scratch is an enormous effort. Today's enterprise applications set strong requirements for concurrency management, security, scalability, robustness, and efficiency. The framework must provide basic services related to these issues, but tackling them successfully is difficult and tedious. Middleware technologies that alleviate the burden of implementing these services from scratch exist. Examples of these kinds of technologies are Microsoft's .NET [Mic02] and Sun Microsystems' J2EE (Java 2 Enterprise Edition) [Sun99]. J2EE is generally recognized as a competitive and mature technology and it is used as the elementary technology for the OpenPort framework discussed in chapter 3. For these reasons, the RDIXML framework is designed to function on any J2EE compatible application server.

The usage of particular J2EE technologies is not further discussed in this thesis, as that would not be interesting considering the goals of this thesis. However, efficient use of J2EE (or any) technology requires profound understanding of how it should be used. It is not just using good technologies that make up a good application or framework; other insights are required for success [Alu01, p. 30]. As Krueger states, it is more difficult to develop reusable solutions than to develop a solution for a specific application as the reusable one is more complex **[**Kru92**]**. The design of the framework should be based on firm and recognized design principles in the chosen technological environment.

A design pattern addresses a recurring design problem and provides a solution to it. Design patterns can be used to document architectural designs. Presenting design ideas with patterns helps the ones implementing the architecture to avoid violating given design ideas [Bus96, p. 6]. In [Alu01], Alur et al. present design patterns that are found to solve common design problems of J2EE-based applications. The next few sections present a set of J2EE patterns that document the key points of the RDIXML architecture, focusing especially on the presentation tier.

Intercepting filter [Alu01, pp. 152-171] – This pattern defines usage of pluggable filters that pre-process and post-process general level basic services concerning requests and responses. The pattern can be used to decorate the main process with filters for example for security, logging, and debugging. The filter mechanism enables adding other services without disturbing the main process. Other duties might include validating user's session and detecting the user's device.

Service to Worker [Alu01, pp. 216-230] **–** This macro pattern combines several other patterns to document the combination of a controller and dispatcher that control views and additional helper classes. The pattern consists of *front controller*, *dispatcher* and *view helper* patterns. The front controller is in charge of invoking system objects based on user request parameters. Dispatcher is responsible of invoking choosing an appropriate (markup specific) view helper for managing the UI processing. The view helper strategy, related to this framework, is further discussed in chapter 5.5.

Composite view [Alu01, pp. 203-215] **–** Views consist of autonomous subviews, which produce the content for a specific layout section of the view. This pattern exists already at the RDIXML language level (in the combination of view, subview, and layoutstructure elements); the framework makes the language level concept concrete by using subview manager objects to complement the view helper pattern.

Business Delegate [Alu01, pp. 248-260] – This pattern provides a single access point to data management tier. It provides an abstraction for and thus hides the implementation of a business service. Hence, it reduces coupling between the presentation tier and the data management tier and thus enhances manageability.

Value Object Assembler [Alu01, p. 339-352] – This pattern is used to compose generic data objects (value objects) from various data sources, i.e. the pattern defines a common mechanism for the communication between the presentation and data management tiers. By forcing each application specific data management object to implement the same interface, the pattern provides an application independent means to deliver data to the presentation tier.

Figure 13 presents the framework's architectural structure in five tiers. The client tier presents the application users that make requests with various kinds of devices. The application logic tier consists of the RDIXML models that are used to build new services. The presentation tier forms the generic part of the framework, which gets its application specific functionality by interpreting the language elements. The data management tier mainly consists of domain objects (EJB beans) that are used to process application specific transactions. The framework tier consists of the J2EE application server, on which the framework is built, and the meta-data that is used to control the technical functionality of the RDIXML framework. The figure also presents the four hot spots that are used to specialize the framework for application specific purposes (see chapter 5.6). Chapter 5.5 provides additional views on the usage of the presented modules and design patterns.

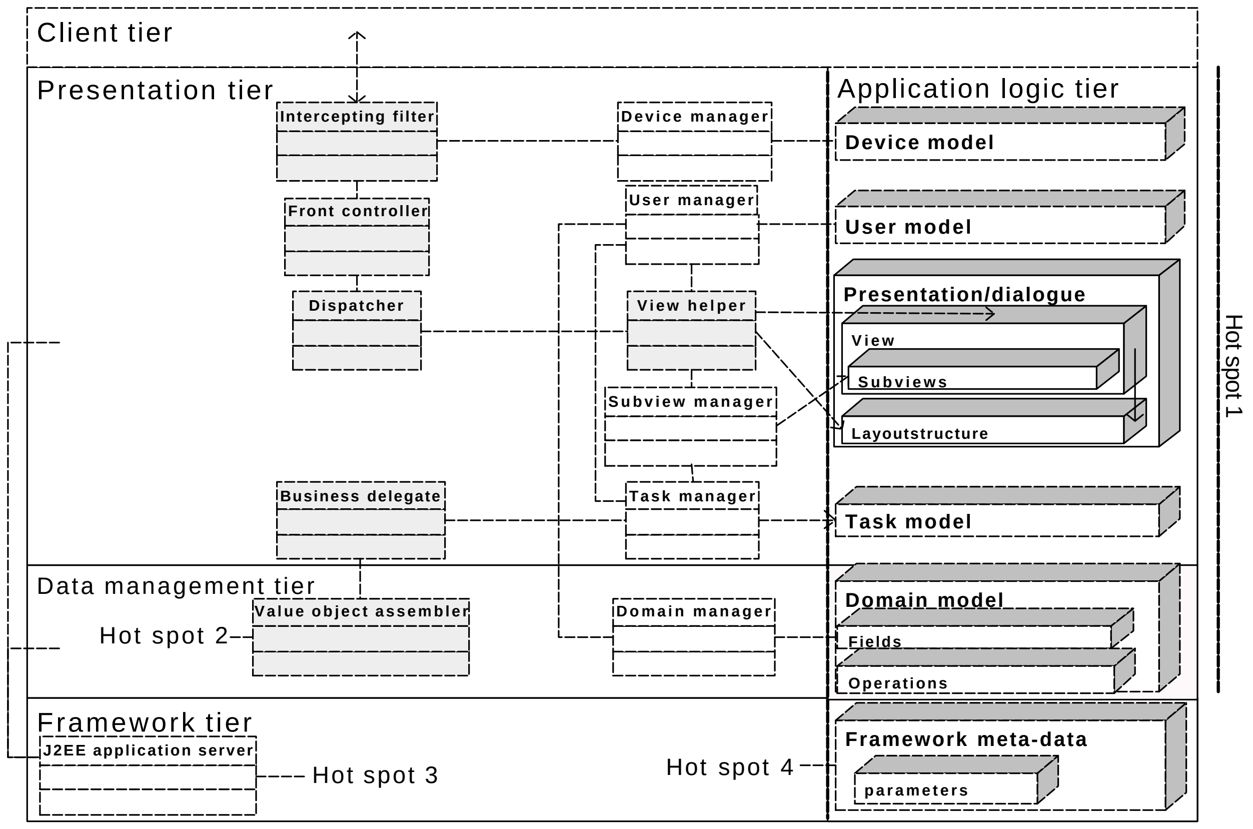


Figure 13: The RDIXML architecture.

The architecture assigns a manager for each of the models of the RDIXML language. This is the key in achieving a generic (application independent) presentation tier, as none of the managers contain any application logic. Each manager is capable of interpreting specific parts of run-time application logic following the RDIXML grammar. For example, view helpers and subview managers dynamically manage the UI by following the instructions given in the presentation/dialogue model.

Due to the task-oriented nature of the framework, design patterns for covering all elementary parts of design do not exist. Hence, the core of the framework cannot purely rely on existing design knowledge. Hence, the adequacy of the given solutions is justified by simulating the functionality of the framework in chapter 6.

5.4 Run-time Application Logic

This chapter gives an overview of issues that relate to constructing run-time data structures from static RDIXML-files and the kind of meta-data that is involved with the generation process. In addition, the techniques and mechanisms used in this process are discussed.

Parsing XML Documents

An XML parser is a software module that is used for reading XML-documents and providing various applications access to their contents and structure [Mar00, pp. 62-64]. The document handling processor can either parse a document in its entirety to a tree structure (that follows the document structure) into computers memory using the DOM parsing model [W3C98], or read the document element by element according to the SAX parsing model [SAX98]. In the latter case, the application that uses the parser has an event driven interface, through which it can apply actions to the elements and attributes of the document as they are being processed. The SAX model enables the programmer to reorganize the parsed elements when necessary. SAX requires less resources compared to DOM, but the application programmer must specifically determine the processing of each element. The DOM model provides a strong tree handling functionality, but may not be the best solution for the manipulation of large XML files [Mar00, p. 183].

Experiences of working with the OpenPort framework (see chapter 3.1) have shown that it is natural to divide application elements to many separate XML-files. It can be assumed that RDIXML applications usually comprise of several files, which all contain partial aspects of the application and share the same application id. This strategy divides the possibly huge amount application data in to smaller units that are easier to control. Many possibilities for the division of elements between files exist. For example, the contents of a single view might be used as a unit for file division. Because several files exist and all of them are not necessarily valid, some kind of meta-data structure is required to tell the system the files to be parsed. The loading order of files might also prove to be important, as some files might contain meta-data of how the following files should be parsed.

An XML-parser is used to load the separate RDIXML-files into a single run-time application logic entity, which is interpreted by other tiers of the framework. Additional research is required to find out, whether this run-time application logic could be optimized to enhance the performance of the framework. Optimizing the model elements of the language for effective access, while still keeping the structures from getting overly redundant, could reduce the required run-time UI adaptation. This would most likely result in better performance, thus enhancing usability. A generally acceptable response time for a user request is 1 seconds. Nielsen states that if the response time exceeds 10 seconds, the user gets frustrated and nearly always moves over to a new site or page [Nie93, Nie97]. For this reason, organizing the run-time application logic matters. Every optimization that can be done to help the dynamic UI creation process will enhance the responsiveness of the server.

For example, a designer might not define presentation for a task, relating to some device group. In this case, the parsing logic could optimize the run-time UI structure by providing a ready-made mapping between UI component types and task attributes. Because there exists several device types, the optimization might require many such mappings. Careful estimations are required concerning performance costs against memory costs in order to find out proper solutions for the run-time structure. If the run-time structure should be reorganized to meet the set requirements, the SAX parsing model should be used instead of DOM, because it provides the means for event-driven element handling. Otherwise, the DOM model might be an appropriate solution, provided that the memory costs would not exceed critical limits.

Validation

An XML parser can automatically check the validity of parsed XML documents referencing a DTD [Mar00, p. 71]. The parser will discard invalid documents and give appropriate error messages accordingly. A DTD is not however a strong enough mechanism for describing the construction rules for all XML applications [Sep02]. For example, ID references cannot be typed. By this, we mean that it is not possible to specify that from inside element “fieldreference”, only to IDs inside elements of type ”field”, can be referred to. In addition, IDs inside a single document are unique independently of their hierarchical element location. Work with the pXML language has proved that this makes it difficult to use rational naming conventions. For these reasons, the RDIXML grammar in appendix A is defined without using XML IDs. The lack of expressive power in DTDs easily leads to situations, where the application is valid on the point of view of the XML parser, but in reality, the application is faulty and useless. Erroneous application logic results in run-time errors, which especially in bigger applications are hard to mend.

In order to guarantee an error free system start-up, the RDIXML engine must provide a separate error management mechanism. The mechanism should be able to detect and report invalid mappings within RDIXML documents. XSLT technology can be used to validate XML-based applications [Sep02]. Another and a more traditional alternative for achieving the required functionality would be to implement a validation module with some programming language. The latter mechanism in combination with additional meta-data information has been used with the OpenPort framework.

The RDIXML language could be implemented with the XML Schema to cover some of the problems concerning application validation. As schema tools are not yet very mature and use of schemas would not solve all mentioned problems [Sep02], schemas are not further discussed in this thesis.

5.5 The RDIXML engine

This chapter presents the fundamental run-time modules of the presentation tier. Technical details, such as session management or device detection, are not discussed; chapter 6 does, however, give an overview of how and when these are managed.

View Management

The UI creation process always happens through view helpers that control the overall layout management of the UI by controlling the content creation and positioning of subviews (see figure 13). On each request, a view helper invokes a subview manager to manage content creation for each of the subviews attached to the view. As a subview manager returns the content, the view helper positions it, renders the structure to device specific markup, and finally outputs the UI.

Each subview manager is responsible of presenting a certain number of tasks and subtasks according to given RDIXML instructions. Each manager thus embeds given task-related content to required UI markup. Subview managers co-operate closely with the task engine to achieve awareness of task states and to get the domain data related to the task (see next section). At a specific moment, some tasks are active, and some are inactive. For the active tasks, subview managers render the task attributes and generate controls through which users can execute actions (defined by “useraction” elements). Rendering static content, such as headings for pages, is also a responsibility of a subview manager.

Subview elements may contain navigational instructions that define new content to be loaded to some part of the view, i.e. the navigational instructions have a reference and a target. The reference defines the content, i.e. what is loaded (which subview), and the target defines the part of the layout into where the content is loaded. A view helper is thus responsible for commanding subview managers to change their content according to dynamically received navigation instructions. According to given layout target, the view helper chooses a subview manager and commands it to replace its content. For example, a subview manager might be commanded to replace “subview A” (current content) with “subview B”. Frankly, a single user action may cause several subviews to be replaced by other subviews. In addition, a view can be loaded to a specific part of the layout. Hence, a view may contain another view, i.e. a view helper may contain another view helper. In these cases, the topmost view helper is responsible of the overall layout management. As the overall layout of the UI changes dynamically, view helpers are responsible of not only managing the UI layout, but also of storing the state of the view as a whole in between user requests.

Task Management

Task management is the basis for nearly all interactivity between the UI engine and user agents. Only static navigational interactions that lead to other services or views are outside the control of the task management system. Task managers control the interactions to both directions thus being decisive elements in the functionality of the UI engine. The overall task status defines the dynamic content shown to the user, even though subview managers function as wrappers for the tasks.

Task manager (see figure 13) maintains each user's comprehensive task status in a user specific task container. The task status predetermines the visibility of data and interaction controls, as subview managers (by default) render only active tasks. Subview managers get the status and content for each of the tasks they contain from the task manager and the output of each task is dependent of its status. On special occasions, a subview definition may define a task to be rendered even if inactive. In these cases, the task is rendered as disabled, i.e. the user sees the task, but is not able to trigger it (see figure 14). By default, inactive tasks are not printed.

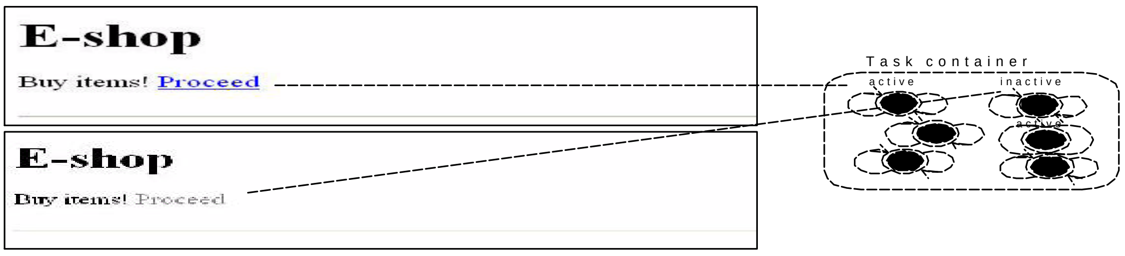


Figure 14: A single task printed as active (above) and as inactive (below).

Completing a task often affects its status. For example, submitting an order is usually a task that the user is only allowed to do once. In addition, completion of a task usually leads to changes in the status of other tasks. Other tasks may be enabled, while others are being disabled. A task may become enabled because some completed task makes a precondition for the disabled task go off. Managing preconditions of tasks seems to require brute force in the sense that all tasks of the application must now be checked for possible enabling. This duty is eased by the hierarchical structure of tasks. Completion of a subtask can only enable subtasks within the on going task. In addition, other comprehensive tasks can be enabled.

A completed task may enable or disable a number of tasks by defining one or more postconditions that can be attached to domain specific keywords. The task engine dynamically matches the keywords returned by the application specific EJB beans to the ones defined by the task description. Each matching keyword triggers a postcondition. In addition to task state changes, postconditions may force the designer to define navigation to another service or task. Due to possible changes, a new comprehensive task status must be computed on the completion of each task. This new task status defines the content to be returned to the user, i.e. the data and interaction possibilities that are rendered to the UI.

From changes in the comprehensive task status it may follow that some subview no longer contains active tasks and if so defined, it will no longer produce output. Subviews that earlier were invisible may now have tasks to present. In addition, navigational instructions may cause new subviews to replace existing ones. Thus it is critical that the designer comprehends managing the overall layout of the UI and provides rational mappings between tasks and presentation. The overall task status must remain such that the layout remains consistent and does not vary in a disturbing manner. Leaving the user without active tasks naturally prevents all interaction.

A task cannot define its execution environment (subview) and thus enabling a task does not necessarily mean that it is presented to the user. If a task is not defined by any of the currently present subviews, a task may be active, but remain invisible. Tasks that are not defined by any of the current subviews can be displayed through navigational instructions. The task model elements can force presentation designers to provide necessary navigational instructions to preserve task flow (see "task model" in chapter 4.2).

User Interface Rendering

Run-time application logic describes applications and their UIs in an abstract format. Once a device makes a request, content must first be adapted according to the device used and the abstract UI definitions must then be transformed into device specific format, to which the content must be embedded. The first phase, i.e. the adaptation functionality, is done by combining the design-time mappings provided by the designer with the dynamic adaptation based on the device profiles (see section "automatic adaptation"). The second phase, i.e. the transformation process, and content embedding, is managed by markup specific transformation modules.

The architecture defines a transformation module for each specific markup language, such as HTML and WML. Each transformation module is aware of the correlation between the abstract UI element and the corresponding markup. Each module is divided into entities capable of constructing markup correlating to given RDIXML elements. Some of these entities must be capable of embedding dynamic values into the markup. For example, a rendering module capable of constructing HTML markup might contain an inner module capable of transforming a text field into HTML format. This inner module uses its reference to a generic data structure (see "value object assembler" in chapter 5.3), extracts the values, and correspondingly outputs '<input type="text" name="name" value="x"/>'. Non-standard browser implementations, especially concerning some specific UI elements, are modelled with device model elements (see "device model" in chapter 4.2). Hence, some transformation entities must consult the device manager for possible deviations from standards.

Due to the generic nature of the tier, the system must use a generic invocation mechanism for the markup specific modules. The invocation targets cannot be hard-coded, since the module to be invoked is dependent on the device detected. Reflection pattern helps changing software's structure and behaviour dynamically [Bus96, p. 193]. Hence, the architecture uses the reflective features of the Java language to dynamically invoke UI environment specific objects according to the device detected.

Figure 15 presents the logical UI rendering process in a very general level. The user's device is the starting point for the rendering process, as it predetermines the process. First, the engine combines the models explicitly mapped by the UI designer. In the absence of design time mappings, the automated mapping mechanism is used (see next section). After that, the device specific transformation module is invoked and given the dynamically retrieved application data. Especially small-scale issues, such as style and size of UI elements, may be affected by the device properties and the module may automatically adapt the presentation according to device requirements. The device specific module transforms the abstract UI into device specific markup, into which it embeds the dynamically retrieved data. Finally, the user interface is sent to the remote device, which uses its rendering capabilities to present a concrete user interface for the user.

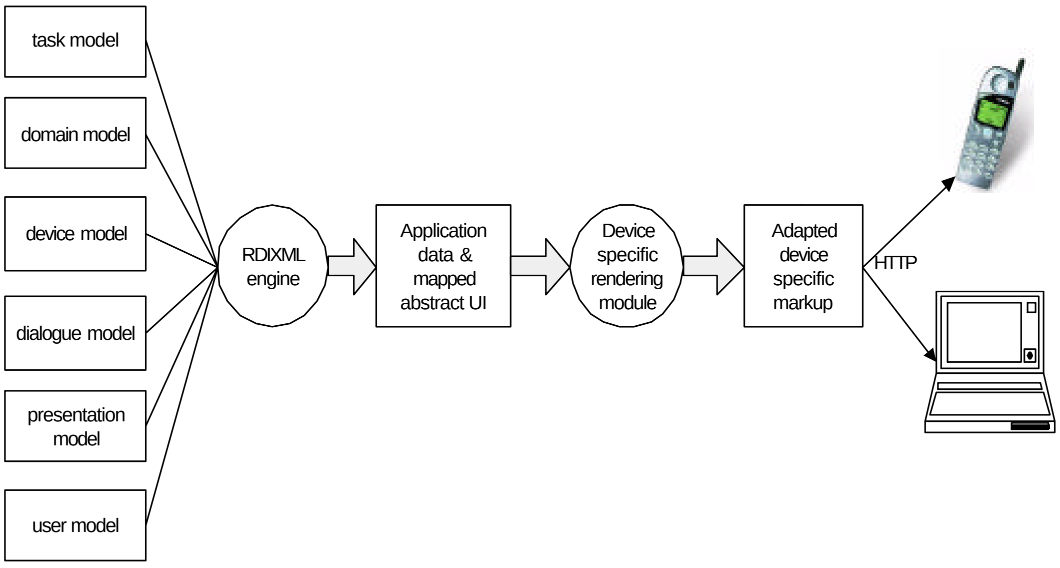

task model
domain model
device model
dialogue model
presentation model
user model
RDIXML engine
Application data & mapped abstract UI
Device specific rendering module
Adapted device specific markup
HTTP

Figure 15: The logical rendering process.

Automatic Adaptation

Designers manage high-level mappings between RDIXML models. By high-level mappings we relate to those mappings that require understanding of the goals considering the application and its usability. For example, it is impossible to automatically define mappings between the task and domain elements. The designer knows what attributes to attach to a specific task and algorithms whatsoever cannot resolve the business specific dependencies. The same applies to what kind of presentation exactly the designer wants to attach to a certain task. For the best end result, the designer must be allowed to rule the overall design process.

There are many things that the system can automatically do, however, to enhance the UI generation process. This automatic adaptation must follow specific rules and guidelines defined by device and markup properties. The simplest prerequisites for automation are the explicitly written abstract-concrete correlations expressed by the device model elements. For example, a designer may choose not to map presentation for task attributes. In this case, the engine can pick the presentation by checking the correlation between the abstract data type and the device type.

On the other hand, the system can use the device properties to infer usage of other model elements. For example, for a device with limited capabilities, the device model may declare that it does not want to use non-obligatory fields. Now, some domain object may define that some attribute is not obligatory, i.e. the value of that attribute is not necessary for the transaction. In this case, the engine can by using these two pieces of information decide to leave those specific attributes out.

The problem in expanding the adaptation functionality is that there are not device specific specifications on how a UI system should react to the device properties. It is not enough to model the properties of the device. It is equally important to describe how the properties of this specific device should be interpreted. This is an obvious problem with the current CC/PP standard, as it does not define such behaviour. In [EIS00], Eisenstein et al. have presented a hierarchical model for describing relations between device properties and possible device specific interaction objects. This could be an interesting way for extending the framework’s device model, as their model clearly opens up views on how UI systems could automatically react to device properties, such as screen resolution. For example, the device model might state that for a certain device with 240x180 resolution the system should use font size 8 and present Boolean values with a checkbox instead of two radiobuttons. However, usability guidelines of this accuracy do not exist and creating them is not a trivial quest. Today’s transformation solutions that are solely based on automation do not provide a credible solution.

There exists another form of mapping in between mappings made by designer and mappings made by the system; we call this *fuzzy mapping*. By this we mean that designers can suggest possible dynamic high-level mappings. An example of this kind of mapping is the case when the designer defines some comprehensive presentation definition (namely the “view” element) as being “default”. In this case, the system maps this presentation for those devices that are not specifically provided one. By making fuzzy mappings, the designer forces the engine to use specific mappings in vague situations. By exploiting the various forms of automatic adaptation, a designer might build simple UIs by implementing the task and domain properties only.

5.6 Specializing the Framework

Frameworks provide ready-made services that can directly be utilized in application development. The users of a framework should never be forced to change the source code of the framework; applications are to be realized by specializing the framework either by combining and configuring existing framework components, or by creating new ones. A framework usually has several specialization points, often called “hot spots”, which are specific to individual applications [Kru96, s. 397]. A framework should clearly define and document the possible specialization mechanisms to avoid a learning curve too steep, thus achieving a higher production level.

On application developers point of view, the RDIXML framework defines four specialization points (see figure 13):

1. The RDIXML language elements
2. EJB beans
3. Application server configuration
4. Framework meta-data

First, several RDIXML files must be written to define all necessary elements of the application and especially the user interface. Secondly, application specific EJB-beans are usually required to implement application specific data processing. Thirdly, the underlying application server technology requires declarations for things like database connections, deployed servlets and EJB-beans, to mention a few. Finally, XML-based meta-data files are used configure the functionality of the RDIXML framework. Examples of meta-data usage are means to control the functionality of the framework's run-time engine, as well as defining relations between language elements for validation purposes.

It would seem that a rational starting point for the UI development is to first define the abstract elements of the UI. Ready-made task and domain implementations provide a solid basis for refining the UI for various devices. Concerning development roles, a domain specialist should see that the database implementation matches the structure of the domain model elements. In addition, the same person should have an understanding of the EJB paradigm. A task analyst could provide initial sketches for task models. Later, co-operation with platform analysts and usability analysts could lead to specializing task flow concerning some devices. In this phase domain and task analysts should work together in order to achieve good quality device specific task models with a good level of domain element reuse. After these preliminary phases, graphical designers together with UI designers can proceed to implement the visual presentation.

User interface languages such as RDIXML or pXML blur the boundaries between architectural layers. These languages at least partly cover issues concerning all three layers of the traditional three-tier architecture model. Hence, new methods for organizing the work and responsibilities are unavoidable. Experience will show how easily new working mechanisms are found and how effectively they can be applied. An example of the difficulties in dividing developer roles relates to presentation: when constructing UI for a regular Web browser, construction of CSS style sheets has commonly been done by graphical designers. However, many PDA devices cannot deal with CSS (e.g. Nokia Communicator). For these kinds of devices, the stylistic issues must be embedded into the RDIXML code.

The objectives of RDIXML, namely related to reuse and adaptability, give rise to its complex modelling structure compared to many other UI languages. It is clear that RDIXML is not as comprehensible to humans, as are some UI languages of more traditional character. This clearly has to do with the use of abstract model elements, which require mappings to separately described UI elements. The complex structure hampers the intelligibility of the UI constructions and directly seems to affect the usability of the RDIXML framework by imposing a need for a tool that would facilitate the UI development process. Editing UI descriptions with regular text editors, or even with XML tools of today’s level, is error-prone and time consuming.

A tool should guide the development process and hide unnecessary details from the designer. First, it should ease the task modelling process. Secondly, it should provide the designer a way to attach domain elements to tasks. After these basic steps, the tool should let the designer see what kind of a default presentation the engine would generate for a given task. Hence, the presentation engine should be integrated with the tool. The designer should be able to browse through device specific views for each task. A working view would allow the designer to map presentation elements to task components. A preview mode would show how that task would be rendered by this certain device according to the current design. This kind of functionality would require that the output of the device specific rendering modules would serve as an input for external user agent simulators.

Despite the above-mentioned problems, interesting possibilities seem to arise from the strict modelling structure of UIM languages, such as RDIXML. As Faulkner and Culwin state, HCI people and software people lack a common vocabulary [Fau00]. Now, UIM languages divide the UI elements into several separate models. Thus, one big topic is divided into several narrower discussion channels with limited context. Instead of having to exchange usability knowledge concerning the whole user interface (which undoubtedly still remains an issue), specific models of the UI can be discussed. This might facilitate achieving consensus of smaller-scale usability guidelines, thus supporting development of usable end-products.

6 E-shop - An Example Use Case

In this chapter, the functionality of the RDIXML framework is put under scrutiny by presenting a use case simulation. During the simulation, the run-time functionality of the framework is examined. At specific points, elements of other models, such as presentation and device, are discussed and sketched. These examples are not provided a grammar. The presented user interfaces have not been designed with aesthetics in mind, rather they are left simple to stress the functional points.

The idea behind the example is to use the framework to realize a single use case for two different devices. A Nokia 9210 Communicator [Nok01a] is chosen to represent a PDA device with a small screen. A regular Web browser (such as Internet explorer or Netscape) used with a high-resolution screen is chosen to represent the average user. In the following chapters, the word "Web browser" always refers to this latter device. The use case is deliberately such that providing both devices a similar flow of actions would not effectively serve the PDA user's needs, i.e. would not provide a usable solution. Hence, the two devices must have diverse sequences of task actions. Different kinds of user interfaces are required in other parts as well, although both devices can render HTML.

6.1 Objectives

The simulation should prove that the RDIXML framework is capable of solving the problems presented in chapter 2.3, i.e. that the framework in practice can produce device dependent task action sequences. To provide a credible simulation, the inner workings of the language and the framework, to some extent, are be presented. Hence, the simulation should reveal any significant weaknesses left unnoticed at the design. Clearly, the simulation should shed some light on what is it like to build applications with the RDIXML framework.

6.2 The Use Case in General

A music wholesaler has an e-shop, through which music-dealers can place orders. Music-dealers can order instruments either by searching or browsing instruments by category. All real world issues that would in practice have to be dealt with are not addressed here. For example, user authorization is not covered. In addition, the users are assumed to have provided necessary billing information earlier. The use case is built around users' activities in buying a product from the shop. The direct flow from the starting point of the service until finishing the order is presented.

Task Model for the Web browser User

Figure 16 presents the task flow of the use case for the user of a regular Web browser. Initially, the user can choose to either hierarchically browse products or to enter a search string. In both cases, if instruments are found, the user can directly add them to shopping cart or browse for remaining search results. After adding products to cart, the user may choose to again browse or search, or may proceed to check out. In the check out phase, the user is viewed the details of the order. From this phase, the user can either accept the order or return for more products.

Task Model for Nokia 9210 Communicator User

The style guide for the Nokia 9210 specifically advises not to use "doormat" pages for this device; it's better to go directly to the service [Nok01b]. The Communicator user pays for the connection all the time, thus speed and efficiency counts and additional browsing should be left for software and equipment that are more sophisticated. Through user interviews, the wholesaler has realized that it is commonly the case that when using a PDA, the user usually just wants to place an order of a certain product and do that quickly. Thus, the service must be optimized for effective ordering.

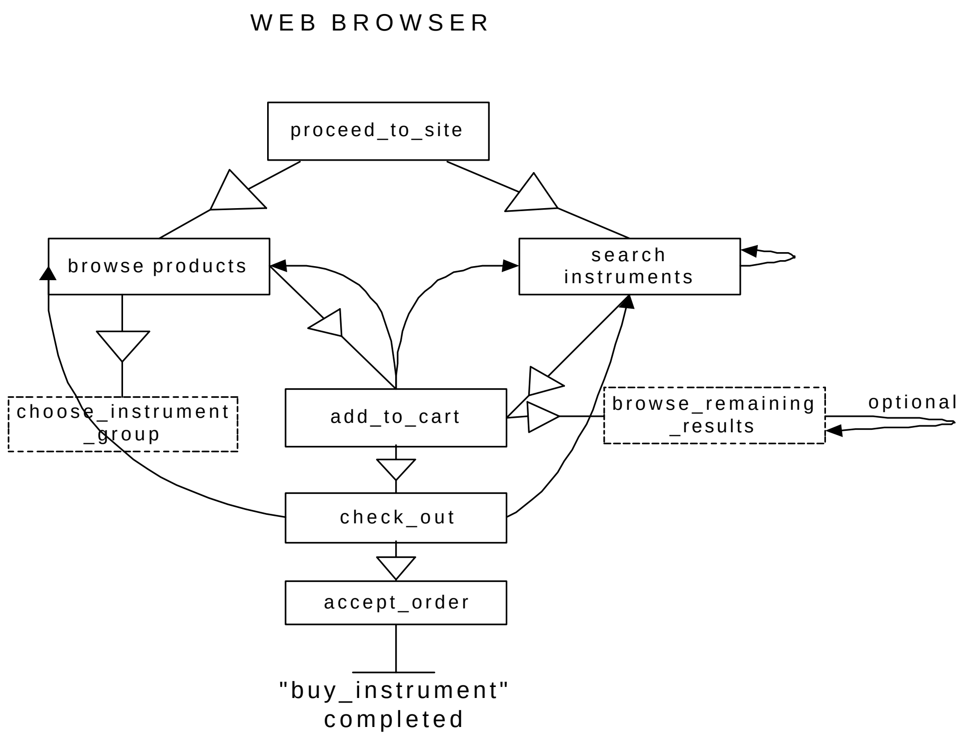


Figure 16: Task flow for WWW-browsers.

The task flow for PDA devices is restricted to one possible route of actions, as can be seen from figure 17. The user can execute a search to find instruments. If instruments are found, the user can either add them to cart or browse for remaining results. If items exist in the cart, the user can directly accept the order. This ought to ease making orders “on the road”. However, the user is provided a possibility to cancel the order within one hour of submitting it. The cancelling process is not described in this use case.

To make the case more realistic, the screen shots in the chapter 6.3 contain some task elements that are neither present in figures 16 and 17, nor in the task model implementation in appendix B.

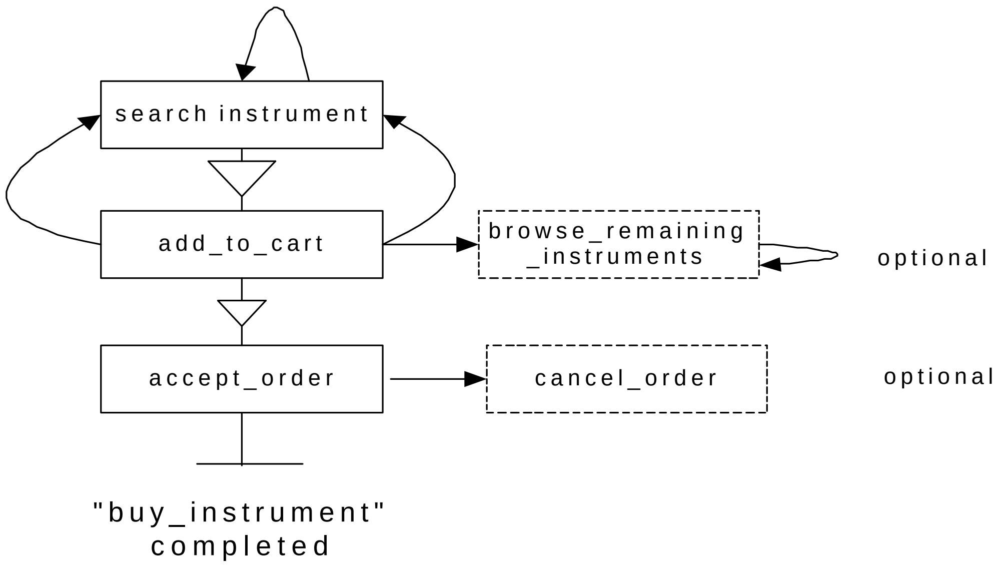


Figure 17: Task flow for PDA devices

6.3 Simulation

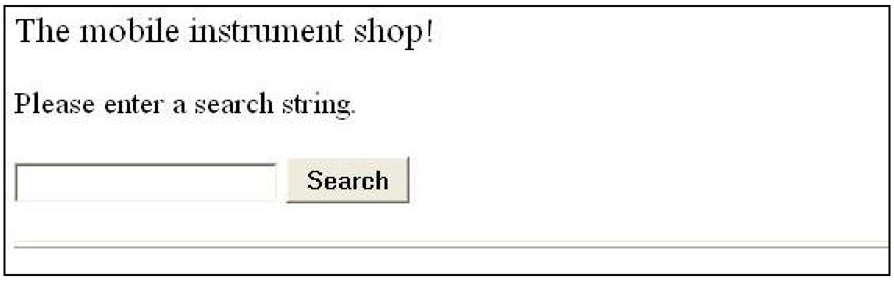
This simulation follows the task description presented in previous chapter. The RDIXML-based task model for both devices is presented in appendix B. Concerning a single subview, the presentation elements for the PDA device are roughly sketched in appendix C. Both appendixes are commented and provide detailed insights to various phases of the simulation.

W3C has published a device independence principle, which states that the URI for a specific service should be the same for all devices [W3C01]. Following this guideline, the framework allows both devices entering the e-shop service by using the same URI. We assume that the RDIXML application model maps the URI to a default view called “shop_view”. As the devices do not share a single starting point for the service, two views named “shop_view” must be implemented, one for the PDA and another for regular Web browsers. Figure 18 outlines the view for the PDA device, where attribute “excludeddevices” excludes Web browsers from this view. Thus, the UI engine is capable of choosing this view for this service each time a PDA device is detected. The view contains two subviews, defined with two “subviewreference” elements. The upper subview contains the welcome text. The lower one, containing the search task, also contains accept order and cancel order tasks, which are both initially disabled.

```
<view view_ID="shop_view" layoutstructure="basic_layout" construction="astable" title="E-shop"
excludeddevices="WWW">
   <subviewreference reference="welcome_subview"/>
   <subviewreference reference="search_subview"/>
</view>
```

Figure 18: Communicator - View structure.

Figure 19 presents the starting point for the PDA users. Since “search_products” is the only task without preconditions, it is the only active task and thus the only one presented.


The mobile instrument shop!

Please enter a search string.

Figure 19: Communicator - Starting point of the service.

Figure 20 presents the layout structure for the view. Attribute “division” is used to define that the subviews are to be set from top to bottom, i.e. that subview “welcome_subview” will be rendered on top of subview ”search_subview”.

```
<layoutstructure layoutstructure_ID="basic_layout" division="vertical" excludeddevices="PDA">
      <frame frame_ID="upper_frame" size="20" dimensiontype="percentage"/>
      <frame frame_ID="lower_frame" size="80" dimensiontype="percentage"/>
</layoutstructure>
```

Figure 20: Communicator - Layout structure for the view.

The sequence diagram presented in figure 21 roughly presents the inner functionality of the framework at the time of the initial request for a service. First, the client makes a request (1). At this stage, the system uses the intercepting filter pattern to tackle several issues related to the request, such as extracting the service name from the URI, detecting the user's device and software, creating an internal (server side) session for the user, and retrieving the name of the default view for the service (2). Relevant parameters, such as the used device and the name of the view name, are now inserted into a parameter structure, which is passed along subsequent calls. This assures that all system objects are able to adapt their output to the request. The intercepting filter now invokes the front controller to delegate control according to the request type (3). The view dispatcher is invoked in (4). Since the Communicator uses the service as a Web browser, an HTML capable view helper is invoked (5).

Following the run-time view presentation, the view helper now invokes two HTML capable subview managers (6) to generate content for both of the RDIXML subview definitions. The subview managers use the task manager to get the status and possible data for both of the tasks that they contain (7). The upper subview definition only contains static text, and thus communication with the task manager is not required. The "search_subview" contains three tasks. Two of them, "accept_order" and "cancel_order" are inactive and not printed. However, the third task, "search_instruments", is active, and thus printed.

At this point, the subview manager does not know the attributes to be rendered, since the subview does not define them. Hence, the task manager must return a list of associated attributes and operations (domain objects mapped to the subtask). Properties such as attribute values and attribute headings are mapped to each attribute in the list. Concerning the search task, the task manager simply returns the abstract data type and label for the search field, and the name and label for the "search" action. The subview manager must now use the automatic mapping process to build the presentation. Hence, the subview manager invokes the device manager to retrieve the presentation type for the attribute and the action (8). Naturally, the device manager needs the current device type and the abstract data types as parameters to be capable of achieving this. As the subview manager receives the presentation types, it can invoke appropriate transformation objects to do final work consisting of creating the markup and embedding the dynamic values into it (9). At this phase, necessary parameters must be bound to the "search" action. As the user triggers the search, the request must then contain information adequate parameters, such as the name of the action and the name of the object that is capable of handling the request. In other words, necessary attributes concerning the RDIXML-based action description must be attached to the user request. At phase (10), the view embeds subview markups into the view markup. Finally, the UI is returned to the client (11).

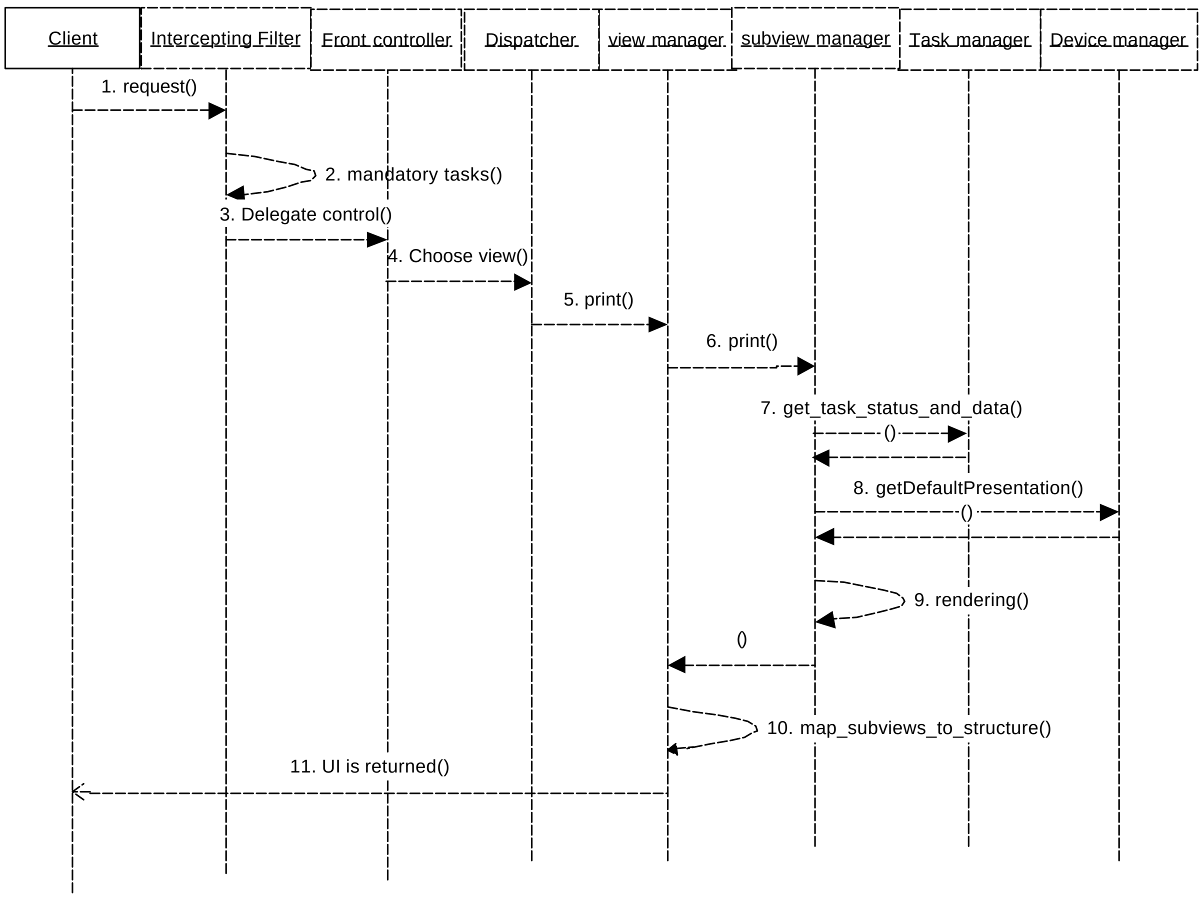


Figure 21: Entering a service.

Users of regular Web browsers enter the service through a flash presentation, which advertises an instrument (see figure 22). This functionality is strictly a part of a business deal made between the wholesaler and instrument manufacturers and has essentially nothing to do with users tasks. Hence, this step is not modelled as part of the task model. This functionality is thus described by using dialogue model elements to provide a link to another view. The “proceed to e-shop” is simply a static link targeted to another view that contains the tasks in question.

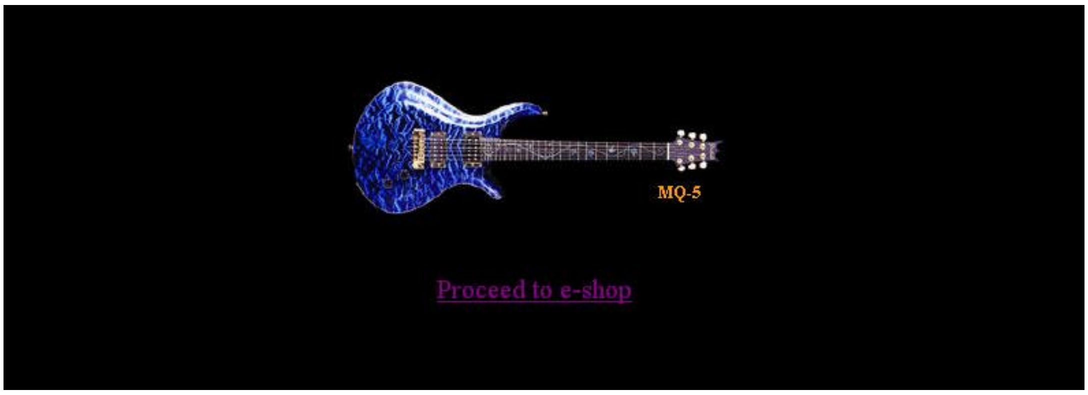
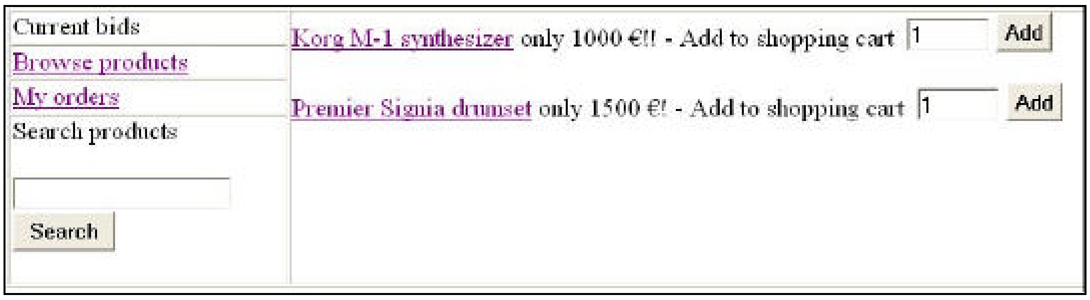


Figure 22: Web browser - A flash advertisement is the user’s route to the service.

Figure 23 presents the starting point of the actual instrument service for the Web browser user. The view initially contains many active subtasks. By default, the user sees the bids of the moment and thus the navigation bar does not show this option as an active link, unlike the others. Other available choices are *Browse products*, *My orders* and *Search products*. Each today’s bid can directly be added to shopping cart. To find items of particular interest, the user can either browse instruments or execute a search.



Korg M-1 synthesizer only 1000 €!! - Add to shopping cart 1

Premier Signia drumset only 1500 €! - Add to shopping cart 1

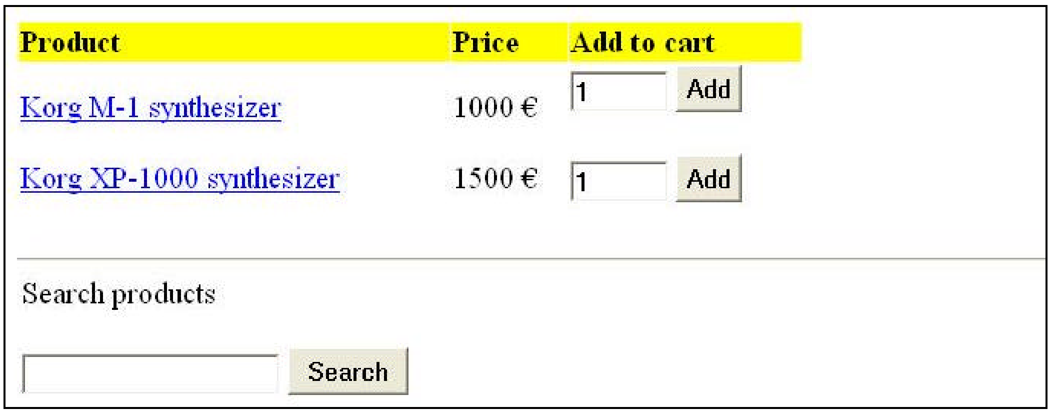
Figure 23: Web browser - Starting point of the service.

The view definition for the Web browser is similar to the one of the PDA. The only differences are that the view refers to different subviews and that it uses a different layout structure. Figure 24 shows that the Web browser layout divides the two subviews vertically, i.e. subviews are presented from left to right.

```
<layoutstructure frameset_ID="shop_layout" division="vertical" excludeddevices="PDA" >
   <frame frame_ID="navigation_frame" size="20" dimensiontype="percentage"/>
   <frame frame_ID="main_frame" size="80" dimensiontype="percentage"/>
</layoutstructure>
```

Figure 24: Web browser - Layout structure for the view.

To proceed the comparison with device interaction, we get back to the PDA user. Following the scenario, the user now enters a search string “korg” and submits the query. At this stage, the task manager executes the query and caches the results to user’s session. The UI engine now follows the navigation properties defined by the occurring postcondition and replaces the “header_subview” with the “add_to_cart_subview”. The “add_to_cart” subtask defines that it uses the data that was cached by the search operation. Figure 25 presents the search results combined with the “add_to_cart” action. The user may now proceed either by submitting a new search, or by adding a certain amount of either item into the shopping cart.



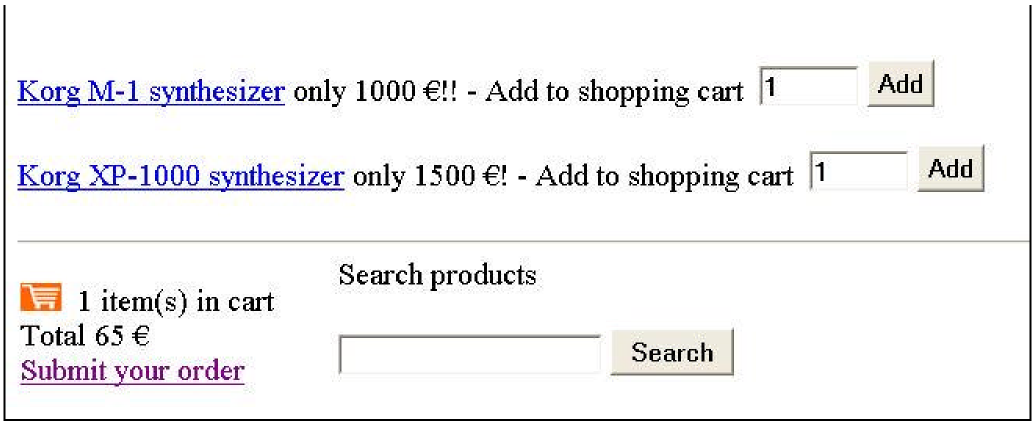
Product	Price	Add to cart
Korg M-1 synthesizer	1000 €	1 Add
Korg XP-1000 synthesizer	1500 €	1 Add

Search products

Search

Figure 25: Communicator - Search results.

The user now triggers the “Add to cart” action. As a result, a product is added to the shopping cart, which is then showed to the user as the result of enabling the “accept_order” task. In figure 26, the user is presented the contents of the cart. While still being able to search for other instruments, the user can now also directly submit the order.



Korg M-1 synthesizer only 1000 €!! - Add to shopping cart 1 Add

Korg XP-1000 synthesizer only 1500 €! - Add to shopping cart 1 Add

1 item(s) in cart
Total 65 €
Submit your order

Search products

Search

Figure 26: Communicator - An instrument is added to the shopping cart.

Figure 27 presents the functionality of the framework in handling the “add_to_cart” action. First, the user triggers the action from the UI (1). The intercepting filter, front controller, and dispatcher patterns are not included in the figure, but they are assumed to function according to their responsibilities. After these basic obligations, the control is delegated to the view helper, which passes the control to the subview manager (2). The subview manager checks if some parts of the referred action description are overridden by the dialogue model and delegates control to the task manager (3).

The task manager executes the task through the business delegate (4). Business delegate uses location services to invoke the “order_handler” EJB bean to complete the task (5). The “session façade” entity presented in the figure is a part of the “value object assembler” design pattern (see figure 13). The name of the business object, as well as the method to be used and the names of the attributes to be retrieved are given to business delegate as call parameters. The EJB bean executes the task and returns a keyword indicating the result of the action. In this case, the operation succeeds and keyword “success” is returned (6). The control now returns to the task manager, which enables task “accept_order” based on the returned keyword (7). Hence, the comprehensive task status of this user changes.

The task manager now returns control to the subview manager, which is informed that the “accept_order” task has been activated. Subview manager returns control to the view helper and based on the given navigational instruction, it informs the view that the “accept_order” task must be rendered by “search_subview”. In phase (8), the view helper ensures that the “search_subview” is among currently active subviews. Since the dialogue model defines that the “accept_order” subtask is to be rendered by the “search_subview”, no subview replacements are required. Since the view helper is not aware of all possible changes in the user’s task status, it now commands each subview manager to refresh their content (9).

In phase (10), subview managers inform the task manager of the tasks (and their status) that they contain. The task manager discovers that the “search_subview” has an inactive task “accept_order”, which has now turned active. It thereby invokes the business delegate to retrieve data as described by the system action “view_cart_data” (11). The business delegate invokes the operation in (12). The subview manager renders the retrieved content (13), as was described in figure 21, and returns the markup to the view helper. Since the view structure does not change, the view need not be refreshed. Finally, the UI is delivered to the user (14).

After submitting the order (figure 28), the PDA user is viewed a confirmation. The user is also given right to cancel the order within one hour of submitting the order.

We now get back to the Web browser simulation. Figure 29 presents the Web users' UI after adding items to the shopping cart. Except for the additional possibilities provided for the Web user, the options to proceed are essentially the same than those of the PDA user. A distinctive feature is, however, that the Web user cannot directly submit the order, but is provided an option to view the contents of the cart by choosing the check out action.

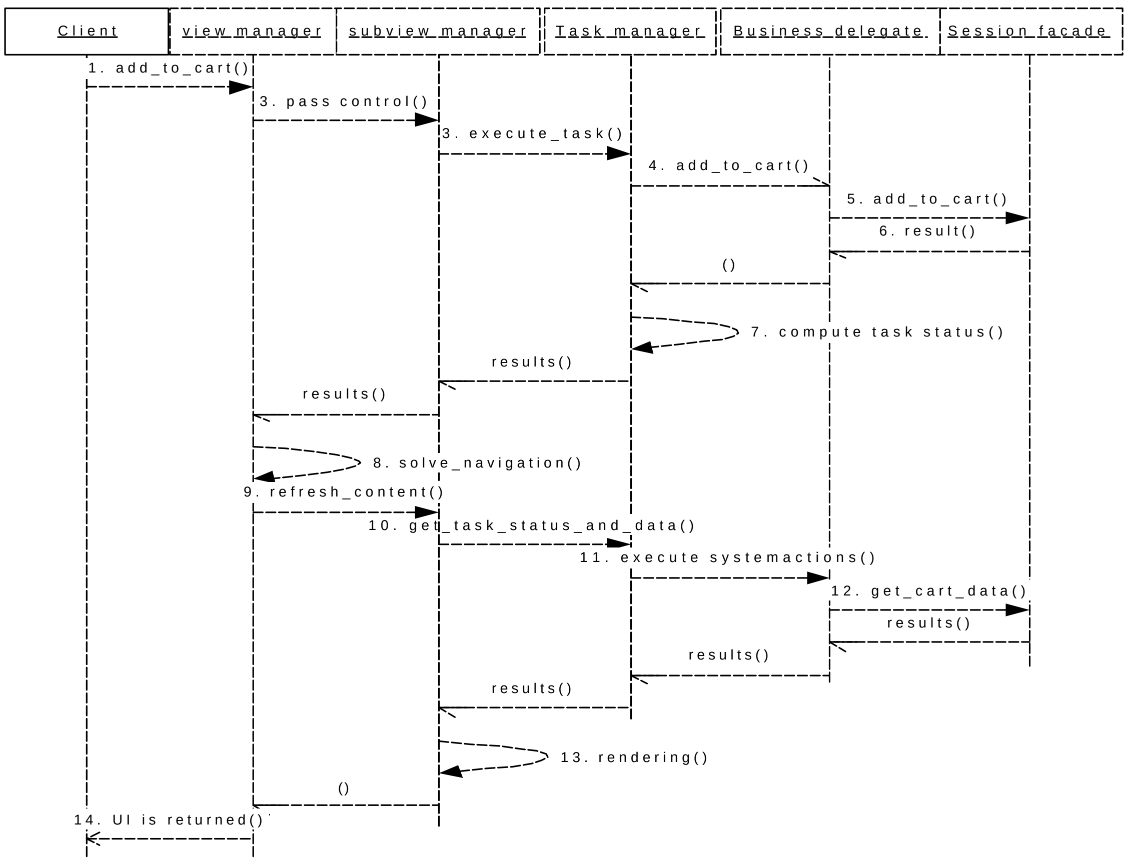
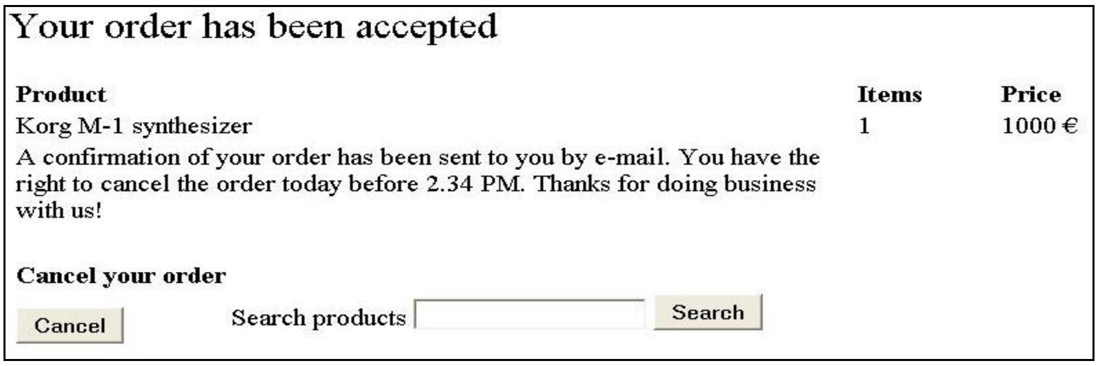


Figure 27: Sequence chart of handling the "add_to_cart" action.



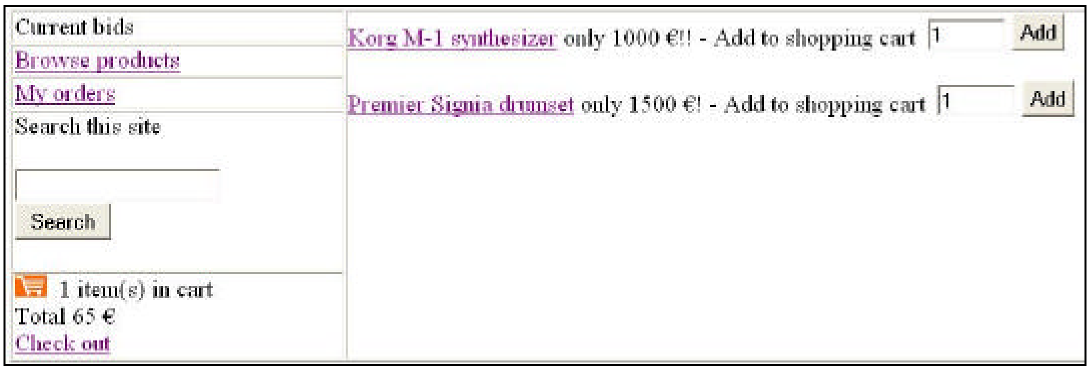
Your order has been accepted

Product	Items	Price
Korg M-1 synthesizer	1	1000 €

A confirmation of your order has been sent to you by e-mail. You have the right to cancel the order today before 2.34 PM. Thanks for doing business with us!

Cancel your order

Figure 28: Communicator - Order accepted.



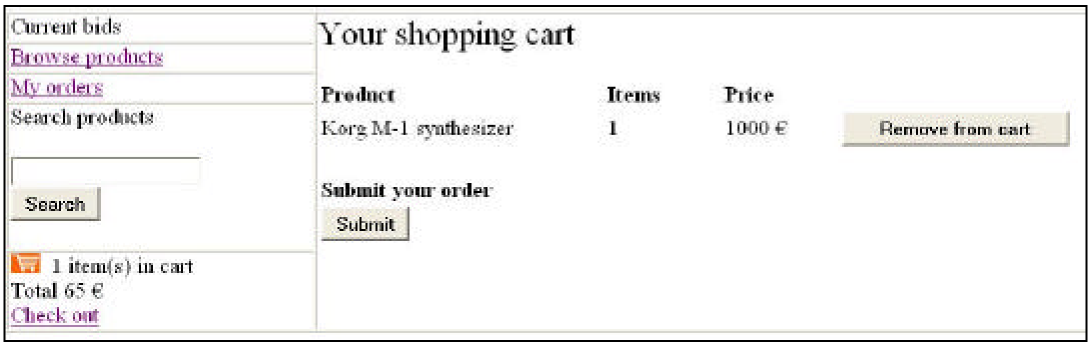
Korg M-1 synthesizer only 1000 €!! - Add to shopping cart 1

Premier Signia drumset only 1500 €! - Add to shopping cart 1

1 item(s) in cart
Total 65 €

Figure 29: Web browser - The contents of the cart.

The user now chooses the check out option and is after this presented the contents of the cart and is provided a possibility to remove items from the cart (figure 30).



Your shopping cart

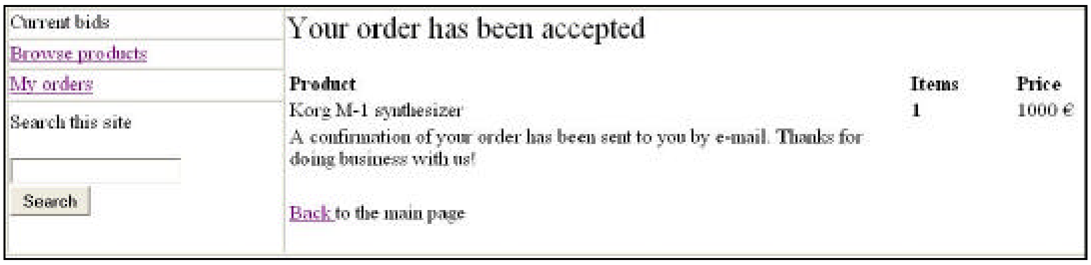
Product	Items	Price	
Korg M-1 synthesizer	1	1000 €	Remove from cart

Submit your order

1 item(s) in cart
Total 65 €

Figure 30: Web browser - The check out phase.

The user now accepts the order and is thus viewed a confirmation (figure 31). The Web browser user is not provided a possibility to cancel the order.



Your order has been accepted

Product	Items	Price
Korg M-1 synthesizer	1	1000 €

A confirmation of your order has been sent to you by e-mail. Thanks for doing business with us!

Back to the main page

Figure 31: Web browser – Order accepted.

7 Analysis

Four main objectives were set for the RDIXML language in chapter 4.1. The first objective, device independence, is satisfied by the characteristics of the language. RDIXML is an abstract language, which does not directly relate to any specific UI environment.

The second objective, adaptation to device specific requirements, is made possible by providing possibilities to map language models to specific devices. The designer has the possibility to design device specific adaptation concerning task, domain, presentation, and dialogue elements. The mappings provide the means to alter the interaction model, domain elements, and the visual or auditory presentation, according to the device detected.

The third objective, task element reuse, is achieved by introducing the concept of DCTM (Device Categorized Task Model), which provides a more fine-grained mapping mechanism between tasks and devices. The concept enables mapping the preconditions, postconditions, and domain elements of tasks to device groups. In addition, reusable interaction descriptions can be referred to from within many separate tasks. These solutions enable devices to reuse common task elements even when some parts of the task are device dependent.

The final objective was to provide a direct integration to the underlying architecture. The language integrates to the underlying RDIXML framework by providing means to describe communication with deployed application objects, i.e. the language lets the designer define attributes and operations that are used to manipulate them. This objective is realized in the OpenPort framework and the RDIXML framework provides an essentially similar concept. In addition, the RDIXML language defines the activation and disabling of tasks that define the interactions between the user and the framework. Both languages move the focus in application development from traditional programming to the use of an abstract level XML-based application language.

In chapter 5.2, four main objectives were set for the RDIXML framework. The first objective, multi-channelled presentation tier is achieved by combing several factors. First, the tier provides a device detection mechanism, which is a prerequisite for content adaptation. Secondly, the tier contains transformation modules for transforming the RDIXML-based UI descriptions to various kinds of target languages, such as HTML. Because RDIXML applications are based on tasks, content can be equally well be transformed to visual and voice-based devices. Finally, the tier uses the reflection pattern for providing a general invocation mechanism, by which device specific functionality can be invoked dynamically. The extent of the tier's communication abilities is dependent of rendering modules, which have to be implemented separately for each markup language.

The second objective, task orientation, was set to provide a fundamentally usability-oriented UI framework and to support goals of pervasive computing. This objective is achieved by binding the visibility of UI content, including interaction possibilities, to the users' comprehensive task status. Designers are enabled to design navigation between tasks and subtasks in a way that forces the engine to maintain a straightforward flow in the task execution.

Third objective, automatic adaptation, is satisfied by exploiting the properties defined by device profiles. First, the abstract-concrete correlations enable the framework to automatically present abstract task elements. Secondly, device properties can be used to infer usage of other model elements. This functionality is in practice difficult to implement because we lack accurate usability guidelines concerning various devices. Thirdly, designers can use fuzzy mappings to force the engine to use specific mappings in vague run-time situations.

Finally, the objective for application independence is achieved by designing a presentation tier such that application logic is wholly defined in another tier. Application development does not require changes in the presentation tier, which gains its functionality by interpreting the run-time application logic.

RDIXML distinguishes from existing UIM languages by introducing the concept of device categorized task model (DCTM). The DCTM solves the dilemma that originates from the notion that device specific requirements in task execution must be addressed in order to achieve usable UI solutions. In other words, task models are not directly portable between devices. Mapping tasks as a whole to certain device groups would significantly reduce the level of UI code reuse. In addition, maintaining information concerning the same task in many locations would provide an error-prone design environment. To achieve a better level of UI code reuse and thus a more systematic development process, the DCTM introduces a fine-grained mapping mechanism between tasks and devices. In essence, DCTM promotes UI code reuse while it still addresses the device-task problem.

The framework's automatic mapping properties support efficient UI development for straightforward cases that do not require additional design. Yet, the automation of the framework does not disarm designers, as the automatic adaptation never overrides definitions made by the designer. The ideology behind the design of this framework specifically stresses the concept of giving the designers the power to design and letting the automatic features provide help without interfering with existing manual design. With its combined properties, the framework strives towards effective development of usable multi-channelled UI solutions without hampering the maintenance of the resulting applications or requiring overly thorough design contribution.

Distinguishing from the rather abstract level academic UIM studies published so far, the RDIXML framework provides a comprehensive usability-oriented concept that uses commercially available standardized technology. Exploiting existing standards is important considering enterprises that are seeking for credible application development models. The academic studies have not presented feasible architectural solutions for task-based multi-channelled application framework; nor have they simulated the functionality of such a framework.

8 Conclusions and Future Work

A subset of RDIXML has been developed as part of this thesis. RDIXML is a model-based user interface language, which sets user's tasks as the starting point for UI development. The language is essentially designed for producing usable multi-channelled applications in a systematic manner. A framework has been presented to implement the RDIXML language. Existing technology has been used in the design, as it was considered important to gain knowledge of the functionality and feasibility of a task-based UI framework. The framework provides functionality to automatically adapt UIs for different kinds of devices and focuses on UI code reuse to support consistent and efficient UI development. The functionality of the framework was successfully simulated in chapter 6.

The main objectives set for the RDIXML language were device independence, adaptation to device specific requirements, task element reuse, and direct integration to the underlying architecture (see chapter 4.1). The main objectives for the RDIXML framework were multi-channelled presentation tier, task orientation, automatic adaptation, and application independence (see chapter 5.2). All objectives for the language and the framework were achieved. An analysis concerning the used mechanisms and observed deficiencies concerning the solutions was given in chapter 7.

A constructive method was used in order to reach given objectives. Detailed knowledge of functional aspects, not to mention design details, of task-based multi-channel UI frameworks has not been published. It would have been very difficult to grasp the related concepts and present a concrete picture of the subject by merely combining existing research material. A better understanding of the overall requirements for a task-based framework was achieved by partially implementing the elements of the language and by simulating the functionality of the framework.

Two paths to proceed from this stage exist. First, the OpenPort framework could be enhanced with the ideas presented in this work. For example, the structure of the pXML language could be clarified. In addition, possibilities for automatic UI generation could be examined based on the thoughts presented in this work. An alternative path is to start developing the RDIXML framework. Although the development would start from scratch, lessons from the OpenPort development provide a technological implementation path from start to finish. Existing pXML language elements, as well as technical parts of the OpenPort framework could be used in the process. The model hierarchy of the language and the task-based approach are the biggest differences that separate the RDIXML framework from OpenPort.

A subset of the RDIXML language has been modelled and principles for the implementation of the other models have been presented. The task model requires some refinement especially concerning relations between tasks. The final form of the device model is unclear mainly due to confusion in benefiting from the CC/PP device model. The rest of the models are merely sketched, but finishing them should be relatively easy. The architecture is now designed in a general level. We estimate that, by reusing portions of the OpenPort framework, a small but skillful team could complete the first version of the RDIXML language and implement a working prototype of the framework within six months. The prototype would include but one device specific rendering module. However, it must be noticed that this thesis does not answer all questions concerning the task model and it is hard to estimate the required work related to open issues.

According to our view, development of the language and the framework should be started in parallel, because the language structures have a significant affect on the functional requirements for the framework. It is difficult to achieve optimal language syntax without practical experiments of its computerized interpretation. A good starting point for the framework development would be to develop prototype implementations of the task and domain models, and the modules for their ma nagement. This point being achieved, issues concerning other models and their management should be easier to tackle. During the development, intensive work should be done considering the various aspects of especially task and device models. Especially relations between tasks are an important issue and explicit rules on how they are expressed must exist. Standardization work done by W3C and other notable instances should be followed, especially concerning the CC/PP standard.

At this stage, setting final goals for the RDIXML framework is not possible, as many aspects of pervasive computing are still developing. New devices and standards keep setting new requirements for the framework, even though the task-based core of the framework can be assumed to remain stable. For example, it is not clear how UIM systems should react to given device model properties. In addition, pervasive computing sets requirements for distributed services. For example, various application servers should be able to exchange information of available services concerning a given device [Ban01].

It is clear that a tool to facilitate the framework usage is necessary. Developing a productivity tool for the system is a difficult and laborious engagement, as the tool should support the UI building process in different phases (see chapter 5.6). Requirements for the productivity tool are numerous and estimations concerning them and the required workload are not provided in this thesis. An example of a rather limited UIM development tool is presented in [Pue98]. An additional aspect that cannot be discussed in the scope of this work, relates to the change of methodology in UI design: how easily will the UI designer community adopt task-based design and the usage of new kinds of user interface languages and models?

We now lack a standardized mechanism to develop usable multi-channel applications. There is a need for a standard that defines an abstract user interface development language with models considering all necessary aspects of a multi-channelled application. The standard would have to include a definition for the functionality of a framework capable of running applications built with the standardized language. This would enable commercial implementations of this standard to free enterprises from the burden of constructing their own frameworks based on several smaller technology standards, such as CC/PP. As such a standard is not being developed, it seems important that the RDIXML framework uses existing standardized technology, such as XML. There are good chances that RDIXML could get advantage from future XML-based smaller standards, as it today it seems probable that if UIM-related standards are published, they will be XML-based.

Finally, it seems that the field of human-computer interaction (HCI) still puts very little effort on research concerning tools and mechanisms that would better support developing usable systems. This work is largely motivated by discussion suggesting that usability research has too much to do with post-evaluation, i.e. that usability research mainly consists of evaluating systems when they are halfway finished. Usability heuristics, which is the main method used for usability evaluation, certainly has its place and value, but usability research excessively emphasizes its usage. Too little emphasis is put on UI languages, methodologies and tools to facilitate systematic development towards usable systems. This is largely due to the lack of both exact definitions for usability and interdisciplinary research between the HCI and software engineering community [Fau00]. This study among others of the similar kind, done with greater consistency and work investment, aimed to show that there are things that can be done.

References

[Alu01] Alur, D., Crupi, J., Malks, D., Core J2EE Patterns – Best Practices and Design Strategies, Prentice Hall, www.phpth.com, [06.03.2002].

[Ban00] Banavar, G. et al., Challenges: An Application Model for Pervasive Computing, In *Proceedings of MOBICOM 2000: the 6th Annual International Conference on Mobile Computing and Networking*, August 6-11, 2000, Boston, MA USA, pp. 266-274.

[Bir98] Birnbaum, L., R. Bareiss, T. Hinrichs, and C. Johnson. 1998. Interface Design Based on Standardized Task Models. In *Proceedings of the 1998 International Conference on Intelligent User Interfaces*. San Francisco, CA: Association for Computing Machinery, pp. 65-72, ACM 1998.

[Bom98] Bomsdorf, B., Szwillus, G., From Task to Dialogue: Task-Based User Interface Design, A CHI'98 Workshop, 1998, call for partition, www.uni-paderborn.de/fachbereich/AG/szwillus/chi99/ws/sigchi.html [06.03.2002].

[Bon99] Bongio, A., Ceri, Stefano, Fraternali, P., Web Modeling Language (WebML): a modeling language for designing Web sites, *Proc. 9th International World Wide Web Conference*, Available at http://www9.org/w9cdrom/177/177.html, Amsterdam, 2000, [06.03.2002].

[Bun01] Buneman, P. et al., Keys for XML, In *Proceedings of 10th International World Wide Web Conference (WWW'10)*, 2001, Available at http://db.cis.upenn.edu/DL/xmlkeys.pdf, [06.03.2002].

[Bus96] Buschmann, F. et al., Pattern-oriented Software Architecture: A System of Patterns, John Wiley and Sons, 1996.

[But01] Butler, M., Current Technologies for Device Independence (External Technical Report HPL-2001-83), Hewlett Packard Laboratories Bristol, http://www.hpl.hp.co.uk/people/marbut/currTechDevInd.htm, [09.03.2002].

[Chu98] Chung-Man Tam, R., Maulsby, D., Puerta, A., U-TEL: A Tool for Eliciting User Task Models from Domain Experts, In *Proceedings of IUI'98, International Conference on Intelligent User Interfaces*, January 6-9, 1998, San Francisco, California, USA.

[Coc97] Cockburn, A., Structuring Use Cases with Goals, JOOP, Vol. 8, No. 6/7 1997, http://members.aol.com/acockburn/papers/usecases.htm, [27.04.2002].

[Con01] Constantine, L., Lockwood, L., Object-modeling and User Interface Design, p. 2, Addison-Wesley, 2001.

[Coo95] Cooper, A., About Face: The Essentials of User Interface Design, p. 389, IDG Books Worldwide, Inc., 1995.

[Cur01] Curious Networks homepage, http://www.curiousnetworks.com, [06.03.2002].

[Eis00] Eisenstein, J., Vanderdonckt, J., Puerta, A., Adapting to Mobile Contexts with User-Interface Modeling, In *Proceedings of the Third IEEE*

Workshop on Mobile Computing Systems and Applications (WMCSA'00), pp. 83-92, 2000.

[Eis01] Eisenstein, J., Vanderdonckt J., Puerta, A., Applying Model-Based Techniques to the Development of UIs for Mobile Computers, In *Proceedings of the 2001 International Conference on Intelligent User Interfaces*, pp. 69-76, January 14-17, 2001, Santa Fe, NM, USA, ACM 2001.

[Elw95] Elwert, T., Schlungbaum, E., Modelling and Generation of Graphical User Interfaces in the TADEUS Approach. In *Proceedings of Designing , Specification and Verification of Interactive Systems,* pp. 193-208, Vienna, 1995. Springer.

[Fau00] Faulkner, X., Culwin, F., Enter the Usability Engineer: Integrating HCI and Software Engineering, In *Proceedings of 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education,* July 11-13, 2000, Helsinki, Finland.

[For99] Forbrig, P., Dittmar, A., Relations between Use Cases and Task Analysis, In *Proceedings of 13 th European Conference on Object-Oriented Programming,* 14-18 June, 1999, Lisbon, Portugal.

[Gri99] Griffiths, T. et al., Teallach: A Model-Based User Interface Development Environment for Object Databases, In *Proceedings of User Interfaces to Data Intensive Systems (UIDIS99),* 5-6th September, 1999, Edinburgh, Scotland, pp. 86-96, IEEE Computer Society Publishers, Norman W. Pator and Tony Griffiths (eds.), 1999.

[Hac98] Hackos, J., Redish, J., User and Task Analysis for Interface Design, Wiley computer publishing, www.wiley.com/compbooks/, [09.03.2002].

[IBM00] Chuvan, C., et al., *The XML Files: Using XML and XSL with IBM WebSphere 3.0*. Available at http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245479.pdf, IBM Corporation, March, 2000.

[Jus01] Juslin, J., Navigation in WAP applications (in Finnish - Navigointi WAP-sovelluksissa), Master's thesis, University of Helsinki, C-2001-35, 2001.

[Kru92] Krueger, C. W., Software Reuse, *ACM Computing Surveys*. Vol. 24, No. 2, June 1992, p. 131-183.

[Lun01] Lundy, J., The Multichannel Content Delivery Opportunity, Research Note, GartnerGroup, 2001.

[Mar00] Martin, D., et al., Professional XML , p. 71, www.wrox.com, [07.03.2002].

[MAX01] Multi-Channel Access XML (MAXML). Available at http://www.curiousnetworks.com/approach.html, Curious Networks, Inc., 2001.

[Mic02] Microsoft .NET homepage, http://www.microsoft.com/net/, [07.03.2002].

[Mül01a] Müller, A., Mundt, T., Lindner, W., Using XML to Semi-automatically Derive User Interfaces. *In Proceedings of 2nd International Workshop on User Interfaces to Data Intensive Systems*, (*UIDIS 2001*), pp. 91-95, IEEE Computer Society, 2001.

[Mül01b] Müller, A., Forbrig, P. and C. H. Cap, *Model-Based User Interface Design Using Markup Concepts*, In *Proceedings of the DSV-IS 2001, Design, Specification and Verification of Interactive Systems*, p. 16 ff., Glasgow, June 2001.

[Mye92] B. A. Myers and M. B. Rosson. Survey on user interface programming. In *Proceedings of SIGCHI'92: Human Factors in Computing Systems*, May 1992.

[Nie93] Nielsen, J., Usability Engineering, Academic Press Inc., 1993.

[Nie97] Nielsen, J., The need for speed (online), Available at http://www.useit.com/alertbox/9703a.html, [12.03.2002].

[Nie00] Nielsen, J., Designing Web Usability: the Practice of Simplicity, p. 38-39, New Riders, 2000.

[Nok01a] The Nokia 9210 Communicator at glance, Available at http://www.nokia.com/phones/9210/, [28.10.2001].

[Nok01b] Nokia 9210 Communicator WWW Browser Style Guide, Version 2, Available at http://www.americas.forum.nokia.com/pointer.asp?url=http://forum.nokia.com/files/disclaimer/1,14553,935,00.html, [04.11.2001].

[Nor86] Norman, D., Draper, S. (editors), User Centered System Design: New Perspectives on Human-Computer Interaction, pp. 87-124, Lawrence Erlbaum Associates Inc., 1986.

[Pue97] Puerta, A., Maulsby, D., Management of Interface Design Knowledge with MOBI-D. In *Proceedings of IUI'97, International Conference on Intelligent User Interfaces,* pp. 249-252, Orlando, FL, January 1997.

[Pue98] Puerta, A., Eisenstein, J., Interactively Mapping Task Models to Interfaces in MOBI-D, In *Proceedings of DSV-IS'98, 5th International Eurographics Workshop on Design, Specification and Verification of Interactive Systems*, pp. 261-273, Abingdon, UK, June 1998.

[Pue99a] Puerta, A., Eisenstein, J., Towards a General Computational Framework for Model-Based Interface Development Systems, In *Proceedings of IUI'99, International Conference on Intelligent User Interfaces*, pp. 171-178, ACM Press, 1999.

[Pue99b] Puerta, A. et al., MOBILE: User-Centered Interface Building, in *CHI99: ACM Conference on Human Factors in Computing Systems. Pittsburgh, May 1999*, ACM press.

[Rat02] The Rational Unified Process (RUP) homepage, http://www.rational.com/products/rup/index.jsp, [07.03.2002].

[San01] Sandnes, F., Self-designing User Interfaces, *In Proceedings of the 27th EUROMICRO Conference: Workshop on Multimedia and Telecommunications*, Warsaw, Poland, IEEE Computer Society Press, September, pp. 452-459, 2001.

[SAX98] SAX, the Simple API for XML 1.0, http://www.megginson.com/SAX/SAX1/index.html, [07.11.2000].

[Sch98] Schlungbaum, E., (Knowledge-based) Support of Task-based User Interface Design in TADEUS, position paper in *CHI'98 Workshop, From Task to Dialogue: Task-based User Interface Design, ACM SIGCHI*

Conference on Human Factors in Computing Systems, April 21-23, Los Angeles, CA, USA, 1998.

[Sch01] Schilit, B. et al., m-links: An Infrastructure for Very Small Internet Devices. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking 2001*, pp. 122-131, July 16-21 2001, Rome, Italy, ACM Press, 2001.

[Sep02] Seppä, A., Dynamic descriptions with XML (in Finnish – Dynaamisuuden kuvaaminen XML:llä), Master's thesis, University of Helsinki, not yet published.

[Shn00] Shneiderman, B., Pushing Human-computer Interaction Research to Empower Every Citizen, Unpublished Manuscript, University of Maryland at College Park, Available at http://www.cs.umd.edu/~ben/p84-shneiderman-May2000CACMf.pdf, [29.04.2002].

[Shn01] Shneiderman, B., Universal Usability as a Stimulus to Advanced Interface Design, A Draft, to Appear in Behaviour & Information Technology 20 th Anniversary Issue.

[Sil00] Silva, P., User Interface Declarative Models and Development Environments: A Survey, In *Interactive Systems: Design, Specification and Verification (7th International Workshop on Design, Specification and Verification of Interactive Systems),* Limerick, Ireland. LNCS Vol. 1946, pp. 207-226, Springer-Verlag, June 2000.

[Sti98] Stirewalt, K., Rugaber, S., Automating UI Generation by Model Composition, In *Proceedings of Automated Software Engineering (ASE'98), 13th IEEE International Conference,* IEEE 1998.

[Sun99] Sun Microsystems, *Java 2 Platform Enterprise Edition Specification, v1.2*, December 1999, Available at http://java.sun.com/j2ee/download.html , [07.02.2002].

[Sun01] Sun Developer Connection, A Framework for Multilingual, Device-Independent Web Sites, April 2001, Available at http://wwws.sun.com/software/mxl/developers/xmlldijsp/framework.html , [29.04.2002].

[Sys01] SysOpen Plc, *Developing Web and Wireless Applications with OpenPor*t. SysOpen Plc, January, 2001, http://www.sysopen.fi.

[Van93] Vanderdonckt, J., Bodart, F., Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection, in *ACM Annual conference on Human Factors in Computing Systems,* pp. 424-429, 1993, ACM Press.

[Van00] [2] Vanderheiden, G., Fundamental Principles and Priority Setting for Universal Usability, *In Proceedings of CUU 2000 - Conference on Universal Usability*, ACM SIGCHI, 2000.

[Voa98] Voas, J., Maintaining Component-Based Systems, *IEEE Softwar*e. Vol. 15, No. 4, July/August 1998, p. 22-27.

[Voi00] VoiceXML Forum, *Voice eXtensible Markup Language (VoiceXML) version 1.0*, May 2000, Available at http://www.w3.org/TR/2000/NOTE-voicexml-20000505/, [07.03.2002].

[W3C98] DOM, Document Object Model, Version 1.0, W3C Recommendation 1 October 1998. http://www.w3.org/TR/REC-DOM-Level-1/, [27.04.2002].

[W3C99a] W3C, *HTML 4.01 Specificatio*n, December 1999, Available at http://www.w3.org/TR/1999/REC-html401-19991224, [07.03.2002].

[W3C99b] W3C, Web Content Accessibility Guidelines (WAI) 1.0, May 1999, Available at http://www.w3.org/TR/WAI-WEBCONTENT/, [07.03.2002].

[W3C99c] W3C, XSL Transformations (XSLT) 1.0, November 1999, Available at http://www.w3.org/TR/xslt, [07.03.2002].

[W3C00a] W3C, *Composite Capabilities/Preference Profiles, (CC/PP)*: Requirements and Architecture, July 2001, Available at http://www.w3.org/TR/2000/WD-CCPP-ra-20000721/, [07.03.2002].

[W3C00b] W3C, *Extensible Markup Language (XML) 1.0 (Second Edition*), October 2000, Available at http://www.w3.org/TR/2000/REC-xml-20001006, [07.03.2002].

[W3C00c] W3C, *Composite Capabilities/Preference Profiles, (CC/PP)*: *Structure and Vocabularies (Working draft)*, March 2001, Available at http://www.w3.org/TR/CCPP-struct-vocab/, [07.03.2002].

[W3C00d] W3C, *XHTML 1.0: The Extensible HyperText Markup Languag*e, January 2000, Available at http://www.w3.org/TR/2000/REC-xhtml1-20000126, [07.03.2002].

[W3C01] W3C, *Device independence Principles* (Working draft 18 September 2001), DIP-2: Device independent Web page identifiers, Available at http://www.w3.org/TR/di-princ/, [17.03.2002].

[WAG01] WAG UAProf, WAP User Agent Profile, Available at http://www1.wapforum.org/tech/documents/WAP-248-UAProf-20010530-p.pdf, [27.04.2002].

[Wah97] Wahl, M., Howes, T., Kille, S., Lightweight Directory Access Protocol (v3), IETF RFC 2251, December 1997, http://www.rfc-editor.org/rfc/rfc2251.txt, [12.03.2002].

[WAP98] Wireless Application Protocol Forum, *Wireless Application Protocol Architecture Specification*, April 1998, Available at http://www1.wapforum.org/tech/terms.asp?doc=WAP-100-WAPArch-19980430-a.pdf [07.03.2002].

[XSL99] XSLT considered harmful, http://www.xml.com/lpt/a/1999/05/xsl/xslconsidered_1.html, [09.03.2002].

[XUL99] XUL (XML-based User Interface Language), Introduction to a XUL-document, Available at http://www.mozilla.org/xpfe/xptoolkit/xulintro.html, [15.10.2001].

A Grammar of RDIXML task model

This appendix contains the commented Document Type Description (DTD) for the RDIXML language.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
DTD FOR RDIXML TASK MODEL GRAMMAR
-->
<!-- A simple entity for a boolean condition -->
<!ENTITY % boolean "(true | false)">
<!-- abstract data types
input - basic input field
input1-n - input field for multiple items
output - basic output field
output1-n - output for multiple fields
-->
<!ENTITY % abstracttype "(input | input1-n | output | output1-n | session_data)">
<!-- ELEMENT DESCRIPTIONS START -->
<!--
ELEMENT application
defines a single application, which contains a number of tasks. In practice, elements from other models should be
included.
-->
<!ELEMENT application (tasks)>
<!--
ELEMENT tasks:
collects several task elements
-->
<!ELEMENT tasks (task+)>
<!--
ELEMENT task:
models a higher lever user goal.

ATTRIBUTES:
task_id - Identifies a task
constructor - a simplified model for describing relations between subtasks
-->
<!ELEMENT task (preconditions*, (subtask | systemtask)*)>
<!ATTLIST task
    task_id CDATA #REQUIRED
    constructor (sequential | repeatable | optional) "sequential"
>
<!--
ELEMENT subtask:
models actions required to complete the higher level task. Subtasks may divide into subtasks.

ATTRIBUTES:
subtask_id - identifies a subtask
constructor - a simplified model for describing relations between inner subtasks
excludeddevices - By default, a subtask is ment for all device types. This element allows excluding devices from the
hierarchical device groups. Reserved word "ALL" can be used to    exclude all devices (so that just some devices can be
included)
includeddevices - If some group is excluded from a subtask, a device included in this hierarchical group can be "re-
included". NOTE: if only "includeddevices" is used, all other device types are excluded from the task!
-->
<!ELEMENT subtask (preconditions*, ((systemaction | systemtaskreference)?, fieldreferences?, useractions?), subtask*)>
<!ATTLIST subtask
    subtask_id CDATA #REQUIRED
    constructor (sequential | repeatable | optional) "sequential"
    excludeddevices CDATA #IMPLIED
    includeddevices CDATA #IMPLIED
>
<!--
ELEMENT navigation
Is used to define places for navigation elements; forces a navigation element definition with dialogue elements -->
<!ELEMENT navigation EMPTY>
<!ATTLIST navigation
    target CDATA #REQUIRED
>
<!--
ELEMENT tasknavigation
Is used to force navigation to another task, i.e to some subview containing the referred task; forces a dialogue reference
to navigate to a subview containing the referred task -->
<!ELEMENT tasknavigation (taskref)>
<!--
ELEMENT preconditions:
Defines preconditions for a task.

ATTRIBUTES:
"excludeddevices" and "includeddevices" - as before
-->
<!ELEMENT preconditions (requiredtasks)>
<!ATTLIST preconditions
    excludeddevices CDATA #IMPLIED
    includeddevices CDATA #IMPLIED
>
<!--
ELEMENT requiredtasks
Defines the required tasks to fulfil a condition. Inner requiredtasks are allowed to specify additional operators (or | and)

ATTRIBUTES
operator - defines relations between required tasks (KUINKA NÄÄ SUHTAUTUU REPEATABLE YMS.. JUTTUIHIN?)
-->
<!ELEMENT requiredtasks (taskref+, requiredtasks?)>
<!ATTLIST requiredtasks
    operator (or | and) #IMPLIED
>
<!-- ELEMENT taskref
Is used to reference tasks
-->
<!ELEMENT taskref (#PCDATA)>
<!--
ELEMENT postconditions
Collects postcondition elements.
-->
<!ELEMENT postconditions (postcondition+)>
<!--
ELEMENT postcondition
Is used to define keywords that EJB objects are committed to use. Discovering that a task is finished is in many cases not
enough. It is important to find out exactly how the task was finished. The output of a task determines its consequences,
i.e. a task may have many implications according to how it was completed. Importantly, post conditions provide a
possibility to enable a task and force navigation to any subtask containing the enabled task.

ATTRIBUTES
keyword - a string returned by a business object must match to this
"excludeddevices" and "includeddevices" - as before
-->
<!ELEMENT postcondition (((enable | disable)?), operation?, (navigation | tasknavigation)?, error?)>
<!ATTLIST postcondition
    keyword CDATA #REQUIRED
    excludeddevices CDATA #IMPLIED
    includeddevices CDATA #IMPLIED
>
<!--
ELEMENT enable
Holds references to those tasks that are enabled due to the postcondition.

ATTRIBUTES
forcenavigation - defines whether the designer has to provide a navigational target for the task (for cases when the task is
not available in the same view
```

```
-->
<!ELEMENT enable (taskref+)>
<!ATTLIST enable
     forcenavigation %boolean;
>
<!--
ELEMENT disable
Holds references to those tasks that are disabled due to the postcondition.
-->
<!ELEMENT disable (taskref+)>
<!--
ELEMENT operation
Describes the operation details by referring to existing domain data

ATTRIBUTES
class - refers to an EJB bean, which must be defined by the domain model
method - refers to an EJB bean's method, which must be defined by the domain model
cachealias - is used to cache retrieved data for later use by other tasks; in practice, other properties, such as time limit for
cache invalidation, would have to be provided.
"excludeddevices" and "includeddevices" - as before
-->
<!ELEMENT operation EMPTY>
<!ATTLIST operation
     class CDATA #REQUIRED
     method CDATA #REQUIRED
     cachealias CDATA #IMPLIED
     excludeddevices CDATA #IMPLIED
     includeddevices CDATA #IMPLIED
>
<!--
ELEMENT error

ATTRIBUTES
type - static: a message defined by the presentation model; method_message: the message returned by the used
business method is used; validation_messages: xxx?
forcenavigation - a separate presentation can be appointed for the message
-->
<!ELEMENT error EMPTY>
<!ATTLIST error
     type (static | method_message | validation_messages) "static"
     forcenavigation %boolean; "false"
>
<!--
This element collects all useractions.
-->
<!ELEMENT useractions (useraction+)>
<!-- ELEMENT useraction:
This element is used to model the interaction flow concerning a triggered UI action. For example, when the user sends a
form to the system, the system most probably wants to pass the data to some business specific object. RDIXML considers
user actions as fully portable at task level. However, dialogue level elements may override task level useractions.

ATTRIBUTES:
useraction_id: The id of the user action is just a name to simply describe the action that the user must make, e.g.
"send_the_form".
-->
<!ELEMENT useraction (operationdescription | navigation | tasknavigation)>
<!ATTLIST useraction
     useraction_id CDATA #REQUIRED
>
<!-- ELEMENT operationdescription
Describes operation properties
-->
<!ELEMENT operationdescription ((operation+ | test), postconditions*)>
<!ELEMENT test EMPTY>
<!--
ELEMENT systemaction:
This element is used to model the actions that the system must do for data retrieval concerning a certain task. In essence,
actions
described inside this element are to be executed before the user actions, i.e. the data that is retrieved through this
information, is provided to the user as a part of the interaction possibility.
```

```
-->
<!ELEMENT systemaction (operationdescription)>
<!--
ELEMENT systemtask
Systemtask provides a reusable way to define systemactions. Many separate subtasks can refer to the same
systemaction for data retrieval purposes.

ATTRIBUTES
systemtask_id - identifies a systemtask
-->
<!ELEMENT systemtask (systemaction?, fieldreferences?)>
<!ATTLIST systemtask
     systemtask_id CDATA #REQUIRED
>
<!-- ELEMENT systemtaskreference
Used by subsystem elements to refer to a systemtask
-->
<!ELEMENT systemtaskreference (#PCDATA)>
<!-- ELEMENT domainreference
Is used to refer to a domain object

ATTRIBUTES
reference - refers to a domain_id
-->
<!ELEMENT domainreference (fieldreference*)>
<!ATTLIST domainreference
     reference CDATA #REQUIRED
>
<!--
ELEMENT fieldreferences
Is used to map domain attributes to a task.

ATTRIBUTES
type - Defines whether the field is an input or an output field. This allows using the device model to automatically choose
default presentation.
            If no fieldreference-elements are found, all attributes of the domain object are printed (according to the role) and
the type is the one given here.
-->
<!ELEMENT fieldreferences (domainreference?)>
<!ATTLIST fieldreferences
     type %abstracttype;
>
<!--
ELEMENT fieldreference:

ATTRIBUTES:
reference - refers to domain model element FIELD
type - Defines whether the field is an input or an output field. This allows using the device model to automatically choose
default presentation.
"excludeddevices" and "includeddevices" - as before
-->
<!ELEMENT fieldreference EMPTY>
<!ATTLIST fieldreference
     reference CDATA #REQUIRED
     type %abstracttype;
     excludeddevices CDATA #IMPLIED
     includeddevices CDATA #IMPLIED
>
```

B E-shop – Task Model Source Code

This appendix contains the source code of the use case’s task model.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE tasks SYSTEM "C:\Opiskelu\Gradu\UIM\Task orientation\task.dtd">
<tasks>
    <task task_id="buy_items" constructor="sequential">
        <!– all tasks are completed in order -->
        <!– BROWSE PRODUCTS - this task can be done continuosly, i.e. it is visible all the time. This task is not
        discussed at the simulation and hence not implemented. -->
        <subtask subtask_id="browse_products" constructor="repeatable" excludeddevices="PDA, WAP"/>
        <!– END OF BROWSE PRODUCTS -->
        <!– SEARCH  INSTRUMENTS  - this task can be done continuosly, i.e. it is visible all the time. In addition, it is
        available for all devices -->
        <subtask subtask_id="search_instruments" constructor="repeatable">
            <fieldreferences>
                <domainreference reference="product_manager">
                    <!-- this field is empty when viewed for the user, so there's no need for "systemoperation" elements.
                    -->
                    <fieldreference reference="search_string"/>
                    <!-- these rest of the attributes are initially not to be rendered, but they must be delivered to the EJB
                   bean. This way the handler of the search action is aware of  which attributes to return of the found
                   items. -->
                    <fieldreference reference="name" type="output"/>
                    <fieldreference reference="price" type="output"/>
                    <fieldreference reference="quantity" type="input"/>
                </domainreference>
            </fieldreferences>
            <useractions>
                <useraction useraction_id="search">
                    <!-- this describes how the triggered "search" action is managed  -->
                    <operationdescription>
                    <!-- operation definition forces the engine to cache the received search results to be used by newly
                    invoked tasks -->
                        <operation class="product_manager" method="search" cachealias="list_instruments"/>
                        <postconditions>
                            <postcondition keyword="items_found">
                                <enable forcenavigation="true">
                                <!-- enable the "add_to_cart" task and force a high-level navigational mapping to the
                                task, i.e. force the designer to present the task immediately. -->
                                    <taskref>add_to_cart</taskref>
                                </enable>
                            </postcondition>
                            <!-- device model may restrict the amount of viewed result rows - this postcondition prepares
                            for that –>
                            <postcondition keyword="cache_manager.items_left">
                                <!-- if all items cannot be viewed at once  -->
                                <enable>
                                    <!-- Let the user browse for more results  -->
                                    <taskref>browse_remaining_instruments</taskref>
                                </enable>
                            </postcondition>
                            <postcondition keyword="no_items_found">
                                <!-- the user is viewed an error message - the presentation model should define how  –>
                                <error type="method_message"/>
                            </postcondition>
                        </postconditions>
                    </operationdescription>
                </useraction>
            </useractions>
        </subtask>
        <!– END OF SEARCH_INSTRUMENTS -->
        <!– ADD_TO_CART -->
        <subtask subtask_id="add_to_cart">
```

```
<preconditions>
    <requiredtasks operator="or">
        <!-- either one of given taskrefs suffice. This condition definition happens to match for both PDA and
       WEB users -->
        <taskref>browse_produc ts</taskref>
        <taskref>search_instruments</taskref>
    </requiredtasks>
</preconditions>
<!-- This system action uses the results that were cached from the "search" method of task "search_items".
-->
<systemaction>
    <operationdescription>
    <!-- note that this reference is user specific and the cache manager gets the data from current user's
    session -->
        <operation class="cachemanager" method="list_instruments"/>
    </operationdescription>
</systemaction>
<fieldreferences>
    <domainreference reference="OrderBean">
        <fieldreference reference="name" type="output"/>
        <fieldreference reference="price" type="output"/>
        <fieldreference reference="quantity" type="input"/>
    </domainreference>
</fieldreferences>
<useractions>
    <useraction useraction_id="submit_to_cart">
        <operationdescription>
            <operation class="order_handler" method="insert"/>
            <postconditions>
            <!-- on successful insertion, the PDA user is provided a possibility to accept the order, and the
            Web user is allowed to proceed to check out. If the cart is already visible, it is refreshed to view
            the correct data (see grammar on "enable"). -->
                <postcondition keyword="success" includeddevices="PDA">
                    <enable forcenavigation="true">
                        <taskref>accept_order</taskref>
                    </enable>
                </postcondition>
                <postcondition keyword="success" includeddevices="WWW">
                    <enable forcenavigation="true">
                        <taskref>check_out</taskref>
                    </enable>
                </postcondition>
            </postconditions>
        </operationdescription>
    </useraction>
</useractions>
<!-- BROWSE REMAINING INSTRUMENTS - this is an inner subtask. Optional inner subtasks are disabled
by default -->
<subtask subtask_id="browse_remaining_instruments" constructor="optional">
    <useractions>
        <useraction useraction_id="browse_remaining_instruments">
            <operationdescription>
                <!-- this is part of the engine's generic functionality -->
                <operation class="cachemanager" method="advance(list_instruments)"/>
                <postconditions>
                    <postcondition keyword="items_left">
                        <enable>
                            <!– This subtask may enable itself. –>
                            <taskref>browse_remaining_instruments</taskref>
                        </enable>
                    </postcondition>
                    <postcondition keyword="default">
                        <!-- all other keywords match to this definition -->
                        <disable>
                            <taskref>browse_remaining_instruments</taskref>
                        </disable>
                    </postcondition>
                </postconditions >
            </operationdescription>
        </useraction>
```

```
          </useractions>
      </subtask>
  </subtask>
  <!– END OF ADD_TO_CART -->
  <!– VIEW_CART_DATA - This is reused by multiple subtasks –>
  <systemtask systemtask_id="view_cart_data">
      <systemaction>
          <operationdescription>
              <operation class="order_handler" method="view_cart_data"/>
          </operationdescription>
      </systemaction>
      <fieldreferences>
          <domainreference reference="order_handler">
              <!-- picture of the cart -->
              <fieldreference reference="cart_picture" type="output"/>
              <fieldreference reference="name" type="output"/>
              <!-- item_count tells the amount of a single item in the order -->
              <fieldreference reference="item_count" type="output"/>
              <!-- items_total is presented beside the cart picture -->
              <fieldreference reference="items_total" type="output"/>
              <fieldreference reference="price" type="output"/>
              <fieldreference reference="total_price" type="output"/>
          </domainreference>
      </fieldreferences>
  </systemtask>
  <!– CHECK OUT - not activated for the PDA users -->
  <subtask subtask_id="check_out" excludeddevices="PDA">
      <preconditions>
          <requiredtasks>
              <taskref>add_to_cart</taskref>
          </requiredtasks>
      </preconditions>
    <!-- This task reuses external systemaction. Since all fields are not needed, the presentation model can leave
    some of them out -->
      <systemtaskreference>view_cart_data</systemtaskreference>
      <useractions>
          <useraction useraction_id="check_out">
              <!-- navigating instruction forces navigation and changes the status of the target task to enabled -->
              <tasknavigation>
                  <taskref>accept_order</taskref>
              </tasknavigation>
          </useraction>
      </useractions>
      <subtask subtask_id="remove_from_cart">
          <!-- Items can be removed from the cart –>
          <useractions>
              <useraction useraction_id="submit_to_cart">
                  <operationdescription>
                      <operation class="order_handler" method="remove"/>
                      <postconditions>
                          <postcondition keyword="ok">
                      <!-- The dialogue model must refresh the current view (stored in the server side session) –>
                              <navigation target="CURRENT_VIEW"/>
                          </postcondition>
                      </postconditions>
                  </operationdescription>
              </useraction>
          </useractions>
      </subtask>
  </subtask>
  <!– ACCEPT ORDER -->
  <subtask subtask_id="accept_order">
      <preconditions includeddevices="PDA">
          <requiredtasks>
              <taskref>add_to_cart</taskref>
          </requiredtasks>
      </preconditions>
      <preconditions includeddevices="WWW">
          <requiredtasks>
              <taskref>check_out</taskref>
```

```
                </requiredtasks>
            </preconditions>
            <!-- External system action definition is reused -->
            <systemtaskreference>view_cart_data</systemtaskreference>
            <useractions>
                <useraction useraction_id="accept_order">
                    <operationdescription>
                    <!-- Although device type information reaches the order_handler object automatically, a different
                    operation is used to manage PDA orders -->
                        <operation class="order_handler" method="accept_order" excludeddevices="PDA"/>
                        <operation class="order_handler" method="accept_order_with_delay"/>
                        <postconditions>
                            <postcondition keyword="error">
                                <error type="method_message"/>
                            </postcondition>
                            <postcondition keyword="ok">
                                <enable>
                                    <taskref>view_confirmation</taskref>
                                    <taskref>navigate_to_start</taskref>
                                </enable>
                            </postcondition>
                            <!-- For a PDA device, the cancel actions is enabled as well -->
                            <postcondition keyword="ok" excludeddevices="ALL" includeddevices="PDA">
                                <enable>
                                    <taskref>cancel_order</taskref>
                                </enable>
                            </postcondition>
                        </postconditions>
                    </operationdescription>
                </useraction>
            </useractions>
            <!-- VIEW CONFIRMATION -->
            <subtask subtask_id="view_confirmation">
                <!-- At this point the framework uses its cache mechanism to retrieve this data -->
                <systemtaskreference>view_cart_data</systemtaskreference>
                <!-- Possibility to cancel for the PDA user -->
                <subtask subtask_id="cancel_order" constructor="optional" excludeddevices="WWW">
                    <useractions>
                        <useraction useraction_id="cancel_order">
                            <operationdescription>
                                <operation class="order_handler" method="cancel_last_order"/>
                            </operationdescription>
                        </useraction>
                    </useractions>
                </subtask>
                <subtask subtask_id="navigate_to_start" constructor="optional">
                    <useractions>
                        <!-- forces to present a navigation -->
                        <useraction useraction_id="navigate_to_start">
                            <!-- the target does not refer to specific view name -->
                            <navigation target="default_view"/>
                        </useraction>
                    </useractions>
                </subtask>
            </subtask>
        </subtask>
        <!-- END OF ACCEPT ORDER -->
    </task>
    <!-- END OF BUY_ITEM TASK -->
</tasks>
```

C E-shop –Presentation Model Source Code

This appendix sketches a presentation for the PDA device. The sketch does not cover all tasks related to the use case.

```
<?xml version="1.0" encoding="UTF-8"?>
<application application_id="e-shop">
    <views>
        <view view_ID="shop_view" frameset="single_frameset" construction="astable" defaultframetarget="_top"
title="E-shop" devicegroup="PDA" layoutstructure="shop_layout" default="false">
            <subviewreference reference="header_subview"/>
            <subviewreference reference="search_subview"/>
        </view>
    </views>
    <subviews>
        <subview subview_ID="header_subview">
            <!-- In the absence of CSS support, we define the text size here -->
            <text size="14">The mobile instrument shop!</text>
        </subview>
        <!– This subview later replaces the "header_subview" -->
        <subview subview_ID="add_to_cart_subview">
            <heading>
                <title>add_to_cart_subview</title>
                <!-- Communicator does not support Javascript - if it would, scripts (for validation, for example) could be
                attached here -->
            </heading>
            <taskproperties>
                <taskproperty reference="buy_items">
                    <subtaskproperties>
                        <subtaskproperty reference="add_to_cart" printIfDisabled="false">
                        <!-- The task is put inside form; the framework generates the "action" attribute of the form at run-
                        time, i.e. forms the URL -->
                            <form target="self">
                                <!-- the new content replaces this subview -->
                                <!-- the content is wrapped inside a table - all four components at the same row -->
                                <table rowelements="4">
                                    <!-- Communicator does not support CSS, so styles must be specified here -->
                                    <tableheading bgcolor="yellow">
                                        <columnheader>Product</columnheader>
                                        <columnheader>Price</columnheader>
                                        <columnheader>Add to cart</columnheader>
                                    </tableheading>
                                    <!-- presentation for fieldproperties is not defined - automatic mapping is used -->
                                    <actionproperties>
                                    <!-- For a Javascript supporting device, this action might trigger a validation script
                                    (introduced at the heading section -->
                                        <actionproperty reference="submit_to_cart" label="Add">
                                            <UIcomponent type="submit"/>
                                      <!-- the dialogue element below defines the subview that is used to present the
                                      "accept_order" task. Note that the task model forces this definition. In this case
                                      the search_subview already contains the accept_order task, so this would not be
                                      necessary -->
                                            <tasknavigations>
                                                <tasknavigation navigation_id="accept_order"
                                                target="search_subview"/>
                                            </tasknavigations>
                                        </actionproperty>
                                    </actionproperties>
                                </table>
                            </form>
                        </subtaskproperty>
                    </subtaskproperties>
                </taskproperty>
            </taskproperties>
        </subview>
        <subview subview_ID="search_subview">
            <heading>
                <title>search_subview</title>
            </heading>
            <!-- now the thing here is to find out whether to map to task model or to dialogue model, or to both models.
            -->
            <taskproperties>
                <taskproperty reference="buy_items">
                    <subtaskproperties>
                    <!-- Initially, the only enabled subtask is search_items. The subtasks are presented similar to the
                   "add_to_cart" subtask and are thus not sketched here. -->
                        <subtaskproperty reference="search_instruments"/>
                        <subtaskproperty reference="accept_order"/>
                        <subtaskproperty reference="cancel_order"/>
                    </subtaskproperties>
                </taskproperty>
            </taskproperties>
        </subview>
    </subviews>
    <layoutstructures>
        <layoutstructure layoutstructure_ID="basic_layout" division="vertical" devicegroup="PDA">
            <frame frame_ID="upper_frame" size="20" dimensiontype="percentage"/>
            <frame frame_ID="lower_frame" size="80" dimensiontype="percentage"/>
        </layoutstructure>
    </layoutstructures>
</application>
```